

USENIX

conference

proceedings

**7th USENIX
Conference on
File and Storage
Technologies**

*San Francisco, CA, USA
February 24–27, 2009*

Sponsored by
The USENIX Association

USENIX
THE OPEN SOURCE COMMUNITY ASSOCIATION

In Cooperation with ACM SIGOPS,
IEEE Mass Storage Systems
Technical Committee (MSSTC),
and IEEE TCOS

Proceedings of the 7th USENIX Conference on File and Storage Technologies

San Francisco, CA, USA February 24–27, 2009

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • http://www.usenix.org

Thanks to Our Sponsors

Platinum Sponsor



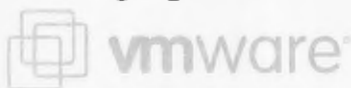
Reception Sponsor



Silver Student Grant Sponsors



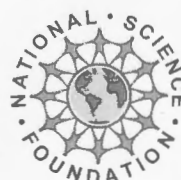
Green Conference Bag Sponsor



Bronze Student Grant Sponsors



Sponsor



Thanks to Our Media Sponsors

ACM Queue

BetaNews

Distributed Management
Task Force, Inc.

InfoSec News

Linux Gazette

Linux Journal

Linux Pro Magazine

LXer.com

Network World ITRoadmap

SNIA

Toolbox.com

© 2009 by The USENIX Association

All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN: 978-1-931971-66-9

USENIX Association

**Proceedings of the
7th USENIX Conference on
File and Storage Technologies**

**February 24–27, 2009
San Francisco, CA, USA**

Conference Organizers

Program Co-Chairs

Margo Seltzer, *Harvard University*
Ric Wheeler, *Red Hat*

Program Committee

Sameer Ajmani, *Google*
Remzi Arpaci-Dusseau, *University of Wisconsin, Madison*
David L. Black, *EMC*
Bill Bolosky, *Microsoft Research, Redmond*
James Bottomley, *Novell*
Daniel Ellard, *BBN Technologies*
Greg Ganger, *Carnegie Mellon University*
Valerie Henson, *Red Hat*
Ethan L. Miller, *University of California, Santa Cruz*
Alina Oprea, *RSA Security/EMC*
James S. Plank, *University of Tennessee*
Calton Pu, *Georgia Institute of Technology*
Raju Rangaswami, *Florida International University*
Narasimha Reddy, *Texas A&M University*
Ohad Rodeh, *IBM Research, Haifa*
Ken Salem, *University of Waterloo*
Jiri Schindler, *NetApp*
Bianca Schroeder, *University of Toronto*
Liuba Shrira, *Brandeis University*
Niraj Tolia, *HP Labs*
Hakim Weatherspoon, *Cornell University*

Work-in-Progress Reports (WiPs) and Poster Session Chair

Geoff Kuenning, *Harvey Mudd College*

Steering Committee

Andrea C. Arpaci-Dusseau, *University of Wisconsin, Madison*
Remzi H. Arpaci-Dusseau, *University of Wisconsin, Madison*
Mary Baker, *HP Labs*
Jeff Chase, *Duke University*
Greg Ganger, *Carnegie Mellon University*
Garth Gibson, *Carnegie Mellon University and Panasas*
Peter Honeyman, *CITI, University of Michigan, Ann Arbor*
Merritt Jones, *MITRE Corporation*
Darrell Long, *University of California, Santa Cruz*
Jai Menon, *IBM Research*
Erik Riedel, *Seagate*
Margo Seltzer, *Harvard University*
Chandu Thekkath, *Microsoft Research*
John Wilkes, *Google*
Ellie Young, *USENIX Association*

The USENIX Association Staff

External Reviewers

Mary Baker
Keith Bostic
Kevin Bowers
John Brainard
Prashanth Bungale
Randal Burns
Ali Butt
Michael Cahill
Gerald Carter
Fay Chang
Peter Chen
Mike Dahlin
David DeWitt
Fred Douglass
Sorin Faibish
Steve Fridella
Mark Gaynor
Garth Gibson
Jason Glasgow
Garth Goodson
Kevin Greenan

David Holland
Galen Hunt
Rebecca Isaacs
Scott Kaplan
Sam Kerner
Mike Kilian
Steve Kleiman
Leonidas Kontothanassis
Orran Krieger
Charles Lamb
Kostas Magoutis
Stephen Manley
Marshall Kirk McKusick
Gerome Miklau
Michael Mitzenmacher
Luc Moreau
Robert Morris
John Niesz
Brian Olson
Shankar Pasupathy
Brian Pawlowski

Avi Pfeffer
Peter Pietzuch
Brian Rogan
Mema Roussopoulos
Steve Schlosser
Himanshu Sinha
Keith Smith
Joe Spiewak
Carl Staelin
Lex Stein
Mark Storer
Sivan Toledo
Mike Ubell
Andy Wang
Matt Welsh
John Wilkes
Jay Wylie
Junfeng Yang
Erez Zadok
Yuanyuan Zhou

7th USENIX Conference on File and Storage Technologies

February 24–27, 2009

San Francisco, CA, USA

Message from the Program Co-Chairs	v
Index of Authors	vi

Wednesday, February 25

Augmenting File System Functionality

The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance	1
<i>Ragib Hasan, University of Illinois at Urbana-Champaign; Radu Sion, Stony Brook University; Marianne Winslett, University of Illinois at Urbana-Champaign</i>	
Causality-Based Versioning	15
<i>Kiran-Kumar Muniswamy-Reddy and David A. Holland, Harvard University</i>	
Enabling Transactional File Access via Lightweight Kernel Extensions	29
<i>Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, and Erez Zadok, Stony Brook University; Charles P. Wright, IBM T.J. Watson Research Center</i>	

Diagnosis

Understanding Customer Problem Troubleshooting from Storage System Logs	43
<i>Weihsiang Jiang and Chongfeng Hu, University of Illinois at Urbana-Champaign; Shankar Pasupathy and Arkady Kanevsky, NetApp, Inc.; Zhenmin Li, Pattern Insight, Inc.; Yuanyuan Zhou, University of Illinois at Urbana-Champaign</i>	
DIADS: Addressing the “My-Problem-or-Yours” Syndrome with Integrated SAN and Database Diagnosis	57
<i>Shivnath Babu and Nedyalko Borisov, Duke University; Sandeep Uttamchandani, Ramani Routray, and Aameek Singh, IBM Almaden Research Center</i>	

Thursday, February 26

Scheduling

Dynamic Resource Allocation for Database Servers Running on Virtual Storage	71
<i>Gokul Soundararajan, Daniel Lupei, Saeed Ghanbari, Adrian Daniel Popescu, Jin Chen, and Cristiana Amza, University of Toronto</i>	
PARDA: Proportional Allocation of Resources for Distributed Storage Access	85
<i>Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger, VMware Inc.</i>	
CA-NFS: A Congestion-Aware Network File System	99
<i>Alexandros Batsakis, NetApp and Johns Hopkins University; Randal Burns, Johns Hopkins University; Arkady Kanevsky, James Lentini, and Thomas Talpey, NetApp</i>	

Tools You Wish You Had

Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality	111
<i>Mark Lillibridge and Kave Eshghi, HP Labs; Deepavali Bhagwat, University of California, Santa Cruz; Vinay Deolalikar, HP Labs; Greg Trezise and Peter Camble, HP Storage Works Division</i>	
Generating Realistic Impressions for File-System Benchmarking	125
<i>Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison</i>	
Capture, Conversion, and Analysis of an Intense NFS Workload	139
<i>Eric Anderson, HP Labs</i>	

Thursday, February 26 (continued)

Metadata and Optimization

- Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems 153
Andrew W. Leung, University of California, Santa Cruz; Minglong Shao, Timothy Bisson, and Shankar Pasupathy, NetApp; Ethan L. Miller, University of California, Santa Cruz
- Perspective: Semantic Data Management for the Home 167
Brandon Salmon, Carnegie Mellon University; Steven W. Schlosser, Intel Research Pittsburgh; Lorrie Faith Cranor and Gregory R. Ganger, Carnegie Mellon University
- BORG: Block-reORGanization for Self-optimizing Storage Systems. 183
Medha Bhadkamkar, Jorge Guerra, and Luis Useche, Florida International University; Sam Burnett, Carnegie Mellon University; Jason Liptak, Syracuse University; Raju Rangaswami and Vagelis Hristidis, Florida International University

Distributed Storage

- HYDRAsTOR: A Scalable Secondary Storage. 197
Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, and Jerzy Szczepkowski, 9LivesData, LLC; Cristian Ungureanu, NEC Laboratories America; Michal Welnicki, 9LivesData, LLC
- Smoke and Mirrors: Reflecting Files at a Geographically Remote Location Without Loss of Performance 211
Hakim Weatherspoon, Lakshmi Ganesh, and Tudor Marian, Cornell University; Mahesh Balakrishnan, Microsoft Research, Silicon Valley; Ken Birman, Cornell University
- Cumulus: Filesystem Backup to the Cloud 225
Michael Vrable, Stefan Savage, and Geoffrey M. Voelker, University of California, San Diego

Friday, February 27

Data Integrity

- WorkOut: I/O Workload Outsourcing for Boosting RAID Reconstruction Performance 239
Suzhen Wu, Huazhong University of Science and Technology; Hong Jiang, University of Nebraska–Lincoln; Dan Feng, Huazhong University of Science and Technology; Lei Tian, Huazhong University of Science and Technology and University of Nebraska–Lincoln; Bo Mao, Huazhong University of Science and Technology
- A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage 253
James S. Plank, University of Tennessee; Jianqiang Luo, Wayne State University; Catherine D. Schuman, University of Tennessee; Lihao Xu, Wayne State University; Zooko Wilcox-O’Hearn, AllMyData, Inc.
- Tiered Fault Tolerance for Long-Term Integrity 267
Byung-Gon Chun and Petros Maniatis, Intel Research Berkeley; Scott Shenker and John Kubiawicz, University of California, Berkeley

Controllers and Caching

- A Systematic Approach to System State Restoration during Storage Controller Micro-Recovery 283
Sangeetha Seshadri, Georgia Institute of Technology; Lawrence Chiu, IBM Almaden Research Center; Ling Liu, Georgia Institute of Technology
- CLIC: CLient-Informed Caching for Storage Servers 297
Xin Liu, Ashraf Aboulnaga, Kenneth Salem, and Xuhui Li, University of Waterloo
- Minuet: Rethinking Concurrency Control in Storage Area Networks 311
Andrey Ermolinskiy and Daekyeong Moon, University of California, Berkeley; Byung-Gon Chun, Intel Research, Berkeley; Scott Shenker, University of California, Berkeley, and ICSI

Message from the Program Co-Chairs

Welcome to the 7th USENIX Conference on File and Storage Technologies! It has been our honor and pleasure to work with an outstanding program committee and the exceptional USENIX staff to bring this program to you.

File and storage technologies continue to be a critical component in our computing landscape—the 102 submissions to FAST this year show just how vibrant and wide-ranging our community of researchers and developers truly is. The 23 papers that will be presented at this conference represent the breadth in storage systems from the home to the cloud.

Winnowing down those 102 submissions to the 23 papers you see here was a monumental task. Our dedicated program committee, in conjunction with 62 external reviewers, produced hundreds of reviews in the six weeks that it took to go through both rounds. In our first round, every submission got at least three reviews. Papers that made it to the second round received a minimum of five reviews, with some getting as many as nine reviews.

The committee met in November for an intense one-day program committee meeting. Every member of the committee was actively engaged in the discussion, and we made a great effort to see that the contribution of every paper was discussed. Differences of opinion were handled professionally, intelligently, and with respect.

Each member of the program committee continued to contribute to the conference after our face-to-face meeting by shepherding at least one paper, helping the authors address the concerns of the reviewers. When USENIX received the final papers, the committee members had to select the best papers and sign up to chair sessions. A simple “thank you” from the program co-chairs seems hardly enough—if you see these people at the conference or encounter them elsewhere, please acknowledge the contribution they made to producing this conference!

We also owe a debt of gratitude to the authors of all the papers submitted. Each paper represented a significant piece of work, and we could have produced multiple, interesting conferences had that been our charter.

This conference could not happen without the outstanding USENIX staff. They are always there to do what needs to get done, nag us into doing what we need to do, solve problems, answer questions, etc. And they always do it with good will and cheer.

And finally, after running conferences for over fifteen years, we were thrilled to have a conference management system that really did what it needed to do. We give a big thank you to Eddie Kohler for his HotCRP system—it’s easy to use and has every feature we needed, and Eddie was more responsive than any commercial customer service organization we’d ever encountered. Many, many thanks!

We hope you enjoy the conference.

Ric Wheeler, Red Hat
Margo Seltzer, Harvard University
FAST ’09 Program Co-Chairs

Index of Authors

Aboulnaga, Ashraf	297	Gulati, Ajay	85	Salem, Kenneth	297
Agrawal, Nitin	125	Hasan, Ragib	1	Salmon, Brandon	167
Ahmad, Irfan	85	Heldt, Lukasz	197	Savage, Stefan	225
Amza, Cristiana	71	Holland, David A.	15	Schlosser, Steven W.	167
Anderson, Eric	139	Hristidis, Vagelis	183	Schuman, Catherine D.	253
Arpaci-Dusseau, Andrea C.	125	Hu, Chongfeng	43	Seshadri, Sangeetha	283
Arpaci-Dusseau, Remzi H.	125	Jiang, Hong	239	Shao, Minglong	153
Babu, Shivnath	57	Jiang, Weihang	43	Shenker, Scott	267, 311
Balakrishnan, Mahesh	211	Kaczmarczyk, Michal	197	Singh, Aameek	57
Batsakis, Alexandros	99	Kanevsky, Arkady	43, 99	Sion, Radu	1
Bhadkamkar, Medha	183	Kilian, Wojciech	197	Soundararajan, Gokul	71
Bhagwat, Deepavali	111	Kubiatowicz, John	267	Spillane, Richard P.	29
Birman, Ken	211	Lentini, James	99	Strzelczak, Przemyslaw	197
Bisson, Timothy	153	Leung, Andrew W.	153	Szczepkowski, Jerzy	197
Borisov, Nedyalko	57	Li, Xuhui	297	Talpey, Thomas	99
Burnett, Sam	183	Li, Zhenmin	43	Tian, Lei	239
Burns, Randal	99	Lillibridge, Mark	111	Treize, Greg	111
Camble, Peter	111	Liptak, Jason	183	Ungureanu, Cristian	197
Chen, Jin	71	Liu, Ling	283	Useche, Luis	183
Chinni, Manjunath	29	Liu, Xin	297	Uttamchandani, Sandeep	57
Chiu, Lawrence	283	Luo, Jianqiang	253	Voelker, Geoffrey M.	225
Chun, Byung-Gon	267, 311	Lupei, Daniel	71	Vrable, Michael	225
Cranor, Lorrie Faith	167	Maniatis, Petros	267	Waldspurger, Carl A.	85
Deolalikar, Vinay	111	Mao, Bo	239	Weatherspoon, Hakim	211
Dubnicki, Cezary	197	Marian, Tudor	211	Welnicki, Michal	197
Ermolinskiy, Andrey	311	Miller, Ethan L.	153	Wilcox-O’Hearn, Zooko	253
Eshghi, Kave	111	Moon, Daekyeong	311	Winslett, Marianne	1
Feng, Dan	239	Muniswamy-Reddy, Kiran-Kumar	15	Wright, Charles P.	29
Gaikwad, Sachin	29	Pasupathy, Shankar	43, 153	Wu, Suzhen	239
Ganesh, Lakshmi	211	Plank, James S.	253	Xu, Lihao	253
Ganger, Gregory R.	167	Popescu, Adrian Daniel	71	Zadok, Erez	29
Ghanbari, Saeed	71	Rangaswami, Raju	183	Zhou, Yuanyuan	43
Gryz, Leszek	197	Routray, Ramani	57		
Guerra, Jorge	183				

The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance

Ragib Hasan
rhasan@illinois.edu
University of Illinois
at Urbana-Champaign

Radu Sion
sion@cs.stonybrook.edu
Stony Brook University

Marianne Winslett
winslett@illinois.edu
University of Illinois
at Urbana-Champaign

Abstract

As increasing amounts of valuable information are produced and persist digitally, the ability to determine the origin of data becomes important. In science, medicine, commerce, and government, data provenance tracking is essential for rights protection, regulatory compliance, management of intelligence and medical data, and authentication of information as it flows through workplace tasks. In this paper, we show how to provide strong integrity and confidentiality assurances for data provenance information. We describe our provenance-aware system prototype that implements provenance tracking of data writes at the application layer, which makes it extremely easy to deploy. We present empirical results that show that, for typical real-life workloads, the runtime overhead of our approach to recording provenance with confidentiality and integrity guarantees ranges from 1% – 13%.

1 Introduction

Provenance information summarizes the history of the ownership of items and the actions performed on them. For example, scientists need to keep track of data creation, ownership, and processing workflow to ensure a certain level of trust in their experimental results. The National Geographic Society’s Genographic Project and the DNA Shoah project (for Holocaust survivors searching for remains of their dead relatives) both track the processing of DNA samples. Individuals who submit DNA samples for testing through these programs want strong assurances that no unauthorized parties will be able to see the provenance of the samples (e.g., provide it to insurance companies or anti-Semitic organizations).

Regulatory and legal considerations mandate other provenance assurances. The US Sarbanes-Oxley Act [56] sets prison terms for officers of companies that issue incorrect financial statements. As a result, officers have become very interested in tracking the path that a financial report took during its development, in-

cluding both input data origins and authors. The US Gramm-Leach-Bliley Act [40] and Securities and Exchange Commission rule 17a [55] also require documentation and audit trails for financial records, as do many non-financial compliance regulations. For example, the US Health Insurance Portability and Accountability Act mandates logging of access and change histories for medical records [13].

Provenance tracking of physical artifacts is relying increasingly on digital shipping, manufacturing, and laboratory records, often with high-stakes financial incentives to omit or alter entries. For example, pharmaceuticals’ provenance is carefully tracked as they move from the manufacturing laboratory through a long succession of middlemen to the consumer. Clinical trials of new medical devices and treatments involve detailed record-keeping, as does US FDA testing of proposed new food additives.

To help manage the above processes, digital provenance mechanisms support the collection and persistence of information about the creation, access, and transfer of data. While significant research has been conducted on how to collect, store, and query provenance information, the associated integrity and privacy issues have not been explored. But without appropriate guarantees, as data crosses application and organization boundaries and passes through untrusted environments, its associated provenance information becomes vulnerable to illicit alteration and should not be trusted.

For example, consider the repudiation incentives in the following *real-life* anonymized medical litigation scenario. Alice visited Dr. B for consultation. B referred her to Dr. Mallory for tests, and sent Alice’s medical records to Mallory, who failed to analyze the test results properly, and provided incorrect information to B. B provided these reports along with other information to Dr. C, who treated Alice accordingly. When Alice subsequently suffered from health problems related to the incorrect diagnosis, she sued B and C for malpractice. To establish

Mallory’s liability for the misdiagnosis, B and C hired Audrey as an expert witness. Audrey used the provenance information in Alice’s medical records to establish the exact sequence of events, which in this case implicated Mallory. If Mallory had been innocent, B and C should not be able to collude and falsely implicate him. Similarly, if Mallory altered his faulty diagnosis in Alice’s medical records after the fact, Audrey should be able to detect that.

Making provenance records trustworthy is challenging. Ideally, we need to guarantee *completeness* – all relevant actions pertaining to a document are captured; *integrity* – adversaries cannot forge or alter provenance entries; *availability* – auditors can verify the integrity of provenance information; *confidentiality* – only authorized parties should read provenance records; and *efficiency* – provenance mechanisms should have low overheads.

In this paper, we propose and evaluate mechanisms for secure document provenance that address these properties. In particular, our *first* contribution is a cross-platform, low-overhead architecture for capturing provenance information at the application layer. This architecture captures the I/O write requests of all applications that are linked with the library, extracts the new data being written and the identity of the application writing it, and appends that information to the provenance chain for the document being written. Further, the resulting provenance chain is secure in the sense that a particular entry in the chain can only be read by the auditors specifically authorized to read it, and no one can add or remove entries from the *middle* of the chain without detection. Our *second* contribution is an implementation of our approach for file systems, along with an experimental evaluation that shows that our approach introduces little overhead at run time, only 1%–13% for typical real-life workloads.

2 Provenance Model

In this section, we define basic provenance-related concepts and discuss deployment and threat models.

2.1 Definitions and Usage Model

A **document** is an abstraction for a data item for which provenance information is collected, such as a file, database tuple, or network packet.

We define **provenance** of a document to be the record of actions taken on that particular document over its lifetime. Note that this definition differs from the information flow provenance used in PASS [38] and some other systems. Each access to a document D may generate a **provenance record** P . The types of access that should generate a provenance record and the exact contents of the record are domain-specific, but in general P may include the identity of the accessing principal; a log of

the access actions (e.g., read, write) and their associated data (e.g., bytes of D or its metadata read/written); a description of the environment when the action was made, such as the time of day and the software environment; and confidentiality- and integrity-related components, such as cryptographic signatures, checksums, and keying material. A **provenance chain** for document D is a non-empty time-ordered sequence of provenance records $P_1 | \cdots | P_n$. In real deployments, the chain is associated and transported together with a document D .

In a given security domain, **users** are principals who read and write to the documents, and/or make changes to document metadata. In a given organization, there are one or more **auditors**, who are principals authorized to access and verify the integrity of provenance chains associated with documents. Every user trusts a subset of the auditors. There can be an auditor who is trusted by everyone, and referred to as the **superauditor**.

Documents, and associated provenance chains are stored locally in the current user’s machine. The local machines of the users are not trusted. Each user has complete control over the software and hardware of her local machine and storage. Documents can be transferred from one machine to another. A transfer of a document from one machine to another also causes the provenance chain to be transferred to the recipient.

Adversaries are inside or outside principals with access to the chains, who want to make undetected changes to a provenance chain for personal benefit. We do not consider denial of service attacks such as the total removal of a provenance chain.

We assume readers are familiar with semantically secure (IND-CPA) encryption and signature mechanisms [22] and *cryptographic hashes* [34]. We use ideal, collision-free hashes and strongly unforgeable signatures. We denote by $S_k(x)$ a public key signature with key k on item x . $a|b$ and a, b denote concatenating b after a .

2.2 Threat Model

In this paper, we focus on tracking document writes and securing the provenance information associated with them. We leave as future work the questions of how to ensure that provenance information is always collected, how to track document reads efficiently, and certain other technical issues discussed below. We now discuss the reasons for choosing this focus.

A provenance tracking system implemented at a particular level is oblivious to attacks that take place outside the view of that level. For example, suppose that we implement provenance tracking in the OS kernel. If the kernel is not running on hardware that offers special security guarantees, an intruder can take over the machine, subvert the kernel, and circumvent the provenance system.

Thus, without a trusted *pervasive* hardware infrastructure and an implementation of provenance tracking at that level, we cannot prevent all potential attacks on provenance chains. Even in such an environment, a malicious user who can read a document can always memorize and replicate portions thereof later, minus the appropriate provenance information. For example, an industrial spy can memorize technical material and reproduce it later on in verbatim or edited form. Overall, it is impractical to assume that we have the ability to fully monitor the information flow channels available to attackers. Thus, our power to track the origin of data is limited.

Fortunately, in many applications of provenance-aware systems, illicit document copying and/or complete removal of provenance chains are not significant threats. For example, in cattle tracking, we are not worried that someone will try to steal a (digital record of a) cow and try to pass it off as their own. Similarly, a cow with no provenance record at all is highly suspect, and many different parties would have to collude to fabricate a convincing provenance history for that cow. Instead, the primary concern is that a farmer might want to rewrite history by omitting a record showing that a particular sick cow previously lived at his feed lot. As another application, a retail pharmacy will not accept a shipment of drugs unless it can be shown that the drugs have passed through the hands of certain middlemen. Thus, if an enterprising crook wants to sell drugs manufactured by an unlicensed company, he might want to forge a provenance chain that gives the drugs a more respectable history, in order to move them into the supply chain. Similarly, there is little danger that someone will remove the provenance chain associated with a box of Prada accessories, and try to pass them off as another brand. Instead, the incentive is to pass off non-Prada accessories as Prada. This is very hard to do, as a colluder with the ability to put the Prada signature on the accessories' provenance chain needs to be found. Anyone who can do that signature is a Prada insider, and Prada insiders have little incentive to endorse fake merchandise.

Of course, there are applications where there are significant incentives to track data reads and for malicious parties to sever documents from their original provenance chains. For example, one may track the flow of intelligence information so that it can be traced back to its original source. It is possible to track all reads and then use that to track information flows. However, doing that is expensive, especially in the long run as provenance chains get longer and longer. It is impossible to offer complete provenance assurances in such situations, but certain measures can be taken, such as digital watermarking and kernel-level tracking of all data read by a writing process [38] (which is expensive due to the need to log all data read). Overall, however, we believe that

in the presence of any non-trivial threat model, tracking read operations for the purpose of provenance collection is impractical and necessarily insecure, because the adversary can always read data through unmonitored channels and produce verbatim or edited copies. Thus, we will not consider the tracking of read operations further in this paper.

The primary threat we guard against in this paper is **undetected rewrites of history**, which occur when malicious entities forge provenance chains to match illicit document writes and metadata modifications. Specifically, suppose that we have a provenance chain $([A], [B], [C], [D], [E], [F])$, in which, for simplicity, each entry is denoted by the identity of its corresponding principal A, B, C, \dots . Then, we will provide the following integrity and confidentiality assurances:

- **I1:** An adversary acting alone cannot selectively remove other principals' entries from the start or the middle of the chain without being detected by the next audit.
- **I2:** An adversary acting alone cannot add entries in the beginning or the middle of the chain without being detected by the next audit.
- **I3:** Two colluding adversaries cannot add entries of other non-colluding users "between" them in the chain, without being detected by the next audit.

For example, colluding users B and D cannot undetectably add entries between their own, corresponding to fabricated actions by a non-colluding party E .

- **I4:** Once the chain contains subsequent entries by non-malicious parties, two colluding adversaries cannot *selectively* remove entries associated with other non-colluding users between them in the chain, without being detected by the next audit.

E.g., colluding users B and D cannot remove entries made by non-colluding user C .

An adversary in possession of a document can always eliminate *all* elements in the chain, starting from the last colluding party's entry in the chain. For example, a malicious F could remove the entry for E , if D cooperates, and claim that the chain is $([A], [B], [C], [D], [F])$. This however, is a denial-of-service attack that cannot be prevented through technical means only. Two parties could always collude in this manner in real life through outside channels. Thus, we target chain forgeries that maliciously add new chain entries and make after-the-fact modifications.

- **I5:** Users cannot repudiate chain records.
- **I6:** An adversary cannot claim that a valid provenance chain for one document belongs to a different document (lineage forgery), without detection at the next audit by a superauditor, if not sooner.

- **I7:** If the adversary alters a document without appending appropriate provenance records to its chain, this will be detected at the next audit by a superauditor, if not sooner.
- **C1:** Any auditor can verify the integrity of the chain without requiring access to any of its confidential components. Unauthorized access to confidential provenance record fields is prevented.
- **C2:** The set of parties originally authorized to read the contents of a particular provenance record for D can be further restricted by subsequent writers of D .

For illustration purposes, consider Bob, Charlie, and Dave editing document D , in that order. The resulting provenance chain contains chronologically ordered entries made by these users. Suppose that, Bob performed an operation on D using a proprietary algorithm, and does not want the workflow he used to be revealed to anyone except auditor Audrey. Property C1 ensures that Bob can selectively reveal the records pertaining to his actions on D to Audrey. Audrey can verify the integrity of the chain and read Bob's action record, while other auditors can only verify the integrity of the chain.

Suppose that Dave subsequently decides to release D to George, who should not learn the private information in Charlie's provenance records. Property C2 allows Dave to give George the provenance chain minus Charlie's sensitive information, while still allowing George to verify the integrity of the chain.

3 A Secure Provenance Scheme

We propose a solution composed of several layered components: encryption for sensitive provenance chain record fields, a checksum-based approach for chain records and an incremental chained signature mechanism for securing the integrity of the chain as a whole. For confidentiality (C1), we deploy a special keying scheme based on broadcast encryption key management [24, 27] to selectively regulate the access for different auditors. Finally, for confidentiality (C2), we use a cryptographic commitment based construction. In the following, we detail these components.

3.1 Building Blocks

3.1.1 Chain Construction

Provenance records (**entries** for short) are the basic units of a provenance chain. Each entry P_i denotes a sequence of one or more actions performed by one principal on a document D :

$$P_i = \langle U_i, W_i, \text{hash}(D_i), C_i, \text{public}_i, I_i \rangle,$$

where

- U_i is an opaque or plaintext identifier for the principal;
- W_i is an opaque representation of the sequence of document modifications performed by U_i ;

- $\text{hash}(D_i)$ is a cryptographic hash of the newly modified contents of D ;
- C_i contains an entry integrity checksum;
- public_i is an optional opaque or plaintext public key certificate for user U_i ;
- I_i contains keying material for interpreting the preceding fields.

As a practical matter, at the start of an editing session, the provenance system should verify that the current contents of D match its hash value stored in the most recent provenance record.

We discuss each of these fields in the following subsections.

3.1.2 Confidentiality

Let w_i be a representation of the sequence of document modifications just now performed by U_i . The choice of representation for w_i is dictated by the application domain; for example, w_i could be a file diff, a log of changes, or a higher-level semantic representation of the alterations. The representation should be reversible, if we want to allow auditors to check whether the current contents of D match its declared history; otherwise the representation does not have to be reversible. Given w_i , W_i is an encrypted version of w_i : $W_i = E(w_i)$.

In the remainder of this section, we discuss options for E that satisfy the C1 confidentiality requirement.

Strawman Choices of E . If all auditors should be able to read all entries, we can encrypt all w_i using a single secret key k shared with the auditors via a central keystore. I.e., $E(w_i) = e_k(w_i)$. If only a subset of the auditors should be able to read w_i , then U_i can encrypt one copy of w_i for each auditor that U_i trusts:

$$E(w_i) = \{e_{K_A}(w_i) : K_A \text{ is the public key of } A, \text{ an auditor } U_i \text{ trusts}\}$$

This is inefficient, as U_i has to include multiple copies of w_i , which may be quite large. One solution approach is to let each auditor in the provenance system correspond to an auditor *role* in the larger environment, and use machinery external to the provenance system to determine who can get access to the auditor role and to track the activities of auditors. Still, the number of different auditor roles can become quite large in a real-world institution. For example, a medical record might be audited by lab technicians, billing people, the patient, her guardians, physicians, and insurers, each with different rights to see details of what was done to the record. Hence, the use of auditor roles external to the provenance system needs to be coupled with internal measures to minimize the size of entries.

To save space, U_i can encrypt w_i with a session key k_i unique to this entry, and also include the (shorter) session key k_i encrypted with the public keys of the trusted

auditors. That is,

$$E(w_i) = e_{k_i}(w_i)$$

$$I_i = \{e_{K_A}(k_i) : K_A \text{ is the public key of } A, \text{ an auditor } U_i \text{ trusts}\}$$

In the rest of this paper, we assume that each entry employs a session key. Still, I_i may have to include many encrypted auditor keys.

Broadcast Encryption for E . To reduce the number of keys that must be included in P_i , we employ broadcast encryption [24, 27]. We illustrate with a specific instance, but any broadcast encryption approach can be used here.

Given a set of up to n auditors during the lifetime of D , we build a broadcast encryption tree of height $\lceil \log N \rceil$. Each leaf corresponds to one auditor, and each node contains a public/private keypair in a PKI infrastructure. We give all the public keys in the tree to every user and auditor. In addition, we give each auditor the private keys in all the nodes on the path from its own leaf to the root.

To allow all auditors to access an entry, we encrypt the session key k_i with the public key in the root of the tree and store it in I_i . Any auditor can then decrypt W_i using the private key in the root node (known only to auditors). Similarly, if U_i trusts a single auditor A , we encrypt k_i with the public key k_A at the leaf for A , so that $I_i = e_{k_A}(k_i)$. If U_i trusts an arbitrary subset S of auditors, we set

$$I_i = \{e_{k_B}(k_i) : B \text{ is the public key of a node in } R\},$$

where R is a minimum-size set of tree nodes such that the set of descendants of R includes all the leaves for auditors in S , and no other leaves. This approach can significantly reduce the size of I_i , as I_i includes only $\log(n - |S|)$ copies of k_i .

If U_i should be opaque, the session key k_i can be used to hide the identity U of the user responsible for the document modifications represented in entry P_i . U should identify the principal in a manner appropriate for the application domain. For example, in a file system we can define U as:

$$U = \langle \text{userID}, \text{pid}, \text{port}, \text{ipaddr}, \text{host}, \text{time} \rangle$$

In an application domain where U_i should be opaque, we can set $U_i = e_{k_i}(U)$, where k_i is the session key defined above. The same can be done for U 's public key certificate public_i . Authorized auditors can use I_i to decrypt U_i and public_i .

In some application domains, we may need finer-grained control over which auditors and users can see which details of an entry, rather than using a single session key to encrypt all sensitive fields in an entry. For

example, we might be willing to let the billing auditors decrypt U_i but not W_i . In this case, one option is to use additional session keys for the other sensitive fields of the entry, so that we can control exactly which auditors can see which fields.

Threshold Encryption. To address separation-of-duty concerns, we can partition the set of auditors into groups, so that decryption of W_i requires joint input from at least one auditor from each group. Alternatively, we can require that at least k different authorized auditors act jointly to decrypt W_i . For these two approaches, we employ secret sharing and threshold cryptography for I_i [49]. Under the first approach, each group has a different share of the session key i_i , and we use the broadcast encryption keys to encrypt those shares. Each auditor can decrypt the share for her group. Under the second approach, there are as many different shares as auditors, and a minimum threshold number of auditors must collaborate to decrypt W_i .

3.1.3 Integrity

Principle **C1** says that every auditor can verify every provenance chain, even if he or she cannot decrypt some of its W_i fields. In some application domains, it is appropriate to allow every user to act as an auditor in this weak sense. In this situation, we can define the integrity checksum field C_i of an entry as:

$$C_i = S_{U_i}(\text{hash}(U_i, W_i, \text{hash}(D_i), \text{public}_i, I_i) | C_{i-1}),$$

where S_{U_i} means that user U_i signs the hash with his or her private key. We refer to this approach as **signature-based checksums**, as it creates a signature chain that enforces the integrity assurances **I1-I7** for each individual entry and for the chain itself. For a user Audrey to be able to verify the chain, she must be able to tell which user wrote each entry in the chain, and have access to their public keys. This can be accomplished by storing the U_i and, if present, public_i fields as plaintext; if the public_i field is empty, Audrey must find the public key for U_i through external means. Audrey can then verify the integrity of the provenance chain by parsing it from beginning to end and using the C_i values to verify the integrity of each entry. She can also verify that the current contents D_n of D match its hash in P_n . However, she cannot check that D_n was computed properly from D_{n-1} unless she is allowed to decrypt W_n , i.e., she is given access to the session key k_n , and a reversible representation was used for w_n . To verify that all of these transformations were performed correctly, Audrey must retrieve her keys from the broadcast encryption tree and use them to decrypt the I_i field of each entry. The I_i field holds the session key for each entry, which she can use to decrypt all of the entry's fields, thus obtaining the w_i fields. From D_n and reversible w_n Audrey can compute D_{n-1} and verify

that it matches its hash in P_{n-1} . Audrey can repeat this process with w_{i-1} , continuing until the entire evolution of D has been verified. If Audrey is not authorized to access all of the session keys for D , then she can only verify that the most recent j entries match the contents of D , where session key k_{n-j} is the most recent session key that she cannot access.

3.1.4 Fine-Grained Control Over Confidentiality

As mentioned earlier, the all-or-nothing approach to allowing auditors to view sensitive fields will be too coarse-grained for some applications. Sometimes it may be hard to foresee which fields may become sensitive over time, especially for a long-lived document that may cross boundaries between organizations. For example, the confidentiality needs for the testing of a particular National Geographic DNA sample may be met perfectly by a particular set of auditors and session keys, as long as the sample stays at its original processing location (the University of Arizona). However, a very small percentage of samples produce ambiguous or seemingly unlikely results (rare genotypes), and these are sent for additional rounds of testing at other labs. When a sample's chain is sent out to a lab in England, the details of previous testing should be eliminated to prevent bias in interpreting the results of the new rounds of tests.

To provide flexibility in such situations without a proliferation of broadcast encryption keys, we can use cryptographic commitments [6] for subfield and field data that may eventually be deemed sensitive. With such a scheme, we can selectively omit plaintext data entirely when sending D 's chain to a new organization, regardless of whether such a need was foreseen when setting up the session key(s) for D . The plaintext information can be restored to the chain if, for example, D later finds its way back to its original organization. To achieve this level of control without a proliferation of encryption keys, we replace each potentially sensitive plaintext subfield s inside U_i or W_i by its commitment before computing the checksum for P_i :

$$\text{comm}(s) = \text{hash}(s, r_s),$$

where r_s is a sufficiently large random number.

During construction of the signature-based checksum, the provenance system uses these hashes instead of the actual data items. For example, the name of an unusual test performed in W_i can be replaced by a commitment, while leaving the other more typical tests in W_i in plaintext. When Arizona sends the chain to an internal party trusted to view the plaintext version of U_i and/or W_i , both the commitments and the original plaintext values of the unusual test s and r_s will be included as usual in the provenance entry. When Arizona sends the chain to

a lab in England, Arizona can remove the plaintext for s and r_s and send only their commitments. Since the chain checksums were computed using the commitments rather than the plaintext data, the English lab can still verify the integrity of the chain. Access to sensitive values is prevented until the chain returns to the University of Arizona, which can reinstate the plaintext in the chain. If the English lab chooses to send out the sample for additional testing, it may choose to omit all the plaintext from all the Arizona entries of the chain. This level of flexibility would be awkward to build into the provenance system using only session keys, but is easily accomplished with commitments.

3.1.5 Augmenting Provenance Chains

While the integrity checksums of Section 3.1.3 completely satisfy the assurances I1-I7, we can introduce further optimizations for faster verification and integrity-preserving summarization of long provenance chains. Provenance chains tend to grow very fast, often becoming several magnitudes in size larger than the original data item and requiring compaction [14]. With our augmented chains, we can compact the chain by removing irrelevant entries, while preserving the validity of integrity verification mechanisms, without requiring recomputing the signatures.

The *integrity spiral*, a redundant, multiple-linked chain mechanism, is conceptually similar to skip-lists [43]. The basic idea is to compute the checksum(s) of each provenance entry by combining the (hash of the) current entry, and multiple previous checksums. We provide two constructions with different properties and usages.

Construction 1. The first construction computes the checksum C_i of the provenance entry P_i as follows

$$C_i = S_{U_i}(\text{hash}(U_i, W_i, \text{hash}(D_i), \text{public}_i, I_i) \\ |C_{prev_1}| \cdots |C_{prev_R}|),$$

where R is the spiral dimension, and C_{prev_k} represents a checksum chosen at random from preceding entries in the provenance chain.

Advantages. This construction allows quick detection of forgery of entries. Suppose that Mallory modifies the entry P_i and computes a new checksum C'_i based on the forged entry and the preceding part of the chain. In the singly-linked mechanism, this will be detected when the auditor checks the checksum for P_{i+1} , which Mallory is unable to forge (per I2). However, the auditor will have to verify the entire chain up to and including the entry P_i to detect this. We enable quick local verification by construction 1, in which multiple subsequent entries will be dependent on the checksum C_i of P_i . The new checksum C'_i added by Mallory will cause the checksums of these dependent entries to fail, and therefore expose

Mallory's forgery. To evade detection, Mallory will have to modify the checksums of all these entries, which in turn will affect further subsequent entries.

Construction 2. In our second construction, we construct a spiral checksum C_i as follows:

$$C_i = C_{i_1} | C_{i_2} | \dots | C_{i_R} | C_{i_0};$$

where R is the spiral dimension, and C_{i_j} is defined as:

$$\begin{cases} S_{U_i}(\text{hash}(U_i, W_i, \text{hash}(D_i), \text{public}_i, I_i) | C_{k_j}) & \text{if } j > 0, \\ S_{U_i}(\text{hash}(U_i, W_i, \text{hash}(D_i), \text{public}_i, I_i) | C_{i_1} | C_{i_2} | \dots | C_{i_R}) & \text{if } j = 0 \end{cases}$$

where $k < i$. Note that, unlike Construction 1, we no longer have a single checksum per entry; rather we use more than one independently computed checksum per entry.

Advantages. Using this construction, we can perform quick verification. The auditor may choose to disregard the linear checksum (i.e. the chain that links to the previous entry's checksum) and use any of the other dimensions. Based on the maximum spiral dimension R of that chain, it might reduce the cost of verification of an N entry chain to $\lfloor \frac{N}{R} \rfloor$. If the chain is constructed such that all entries belonging to a particular event type are linked by a given dimension of the checksum, then the auditor can skip irrelevant entries, but still be able to verify the integrity of the events she is looking for.

Rather than choosing previous checksums randomly, we can use a systematic approach towards building the spiral. For example, to construct C_i , we use checksums from previous entries at distance $1, 2, \dots, 2^R$. In Construction 1, these checksums are all concatenated to the hash of the current entry, while in Construction 2, these R checksums are separately concatenated with the hash of current entry, and signed to form a collection of R checksums pertaining to the current entry.

Integrity-preserving Summarization. Using construction 2, we can compact a provenance chain while still being able to preserve integrity verification mechanism. If P_j occurs after entry P_i in the chain, and P_j 's set of checksums C_j has a checksum C_{j_k} computed from a checksum C_{i_k} from the set C_i (using Construction 2), then that checksum can be used to verify the order of these two entries. If there are d entries between P_i and P_j , we can then remove these entries, while being able to prove the order. This is not contrary to I1, as the auditor can use C_{j_k} to verify the order of P_i and P_j , while detecting that d entries have been removed in between them. If P_i and P_j are not directly connected by a checksum, but have an intermediate entry P_m , such that a checksum exists in C_m that proves P_m occurs after P_i , and a checksum exists in C_j that proves P_j occurs after P_m , then we can keep P_m and remove all other entries between P_i and P_j during compaction. We can extend this technique to link any two entries using some intermediate nodes between them.

3.1.6 Chain Operations

As discussed in Section 2.2, we are concerned with tracking data write operations and document metadata changes – including changes in permissions and other document metadata. We now discuss the impact of document operations on the corresponding secured provenance chains. Although our discussion refers specifically to file system operations, the same semantics are appropriate for other scenarios, including relational databases.

read No impact on the provenance chain.

write A new provenance chain entry is created.

chmod or chown These operations change document metadata. For example, chown can be used to add a new user to the list of users with write access. The change in metadata is recorded as a provenance event.

copy A duplicate copy of the original document is created, with no change in the original document or its provenance. The original document's provenance chain is copied into the new document's provenance chain. A new entry is then added to the new chain, to record the copy operation itself.

delete The document will be removed from the file system, but its provenance chain is *not* deleted. The delete operation is recorded as a metadata operation in the provenance chain. The chain is kept until it expires (determined by its expiration timeout).

If users were guaranteed not to circumvent the provenance-aware read operation (e.g., by reading directly from disk), we could support read-related provenance entries by persisting per-principal provenance contexts containing all information ever read, similar in nature to propagated access list mechanisms [58]. However, as discussed in Section 2.2, we believe that this would give the illusion of security but not the reality, because principals have access to outside information channels. Also, promulgation of provenance for read operations tends to result in a combinatorial explosion in overhead that can ultimately render the system unusable [38].

3.2 Correctness

The mechanisms introduced above satisfy the integrity, confidentiality and privacy properties outlined in Section 2.

Theorem 1. *An adversary cannot remove entries from the beginning or the middle of the chain without detection (I1). An adversary cannot add entries in the beginning or the middle of the chain without detection (I2).*

Proof. (sketch) The proof is straightforward. For (I1) let us assume that an adversary Mallory has removed the entry P_i , $0 < i < n$. Since the integrity checksum field C_{i+1} of the subsequent entry is computed by combining

the current checksum C_i with W_{i+i} under an ideal cryptographic hash function, its verification will fail, therefore revealing the removal of P_i . Similarly, for (I2), any addition of chain entries will be detected in the verification step through the checksum components. \square

Theorem 2. *Once the chain contains any subsequent entries by non-malicious users, a set of colluding adversaries cannot insert or remove entries between them in the chain (I3, I4).*

Proof. (sketch) The chained nature of the integrity checksum directly ensures this. Specifically, suppose that Eve and Mallory are two colluding adversaries who are part of the chain, with entry P_e followed by P_m later on in the chain. Moreover, let P_a be non-colluding Alice's entry following Eve's and Mallory's. The chain will thus be $\langle \dots P_e, \dots, P_m, \dots, P_a, \dots \rangle$. Due to the collision-free, one-way nature of the chained integrity checksum fields, any modification in the chain entries between P_e and P_m will naturally show up when attempting to verify P_a 's checksum. \square

As discussed in Section 2, if P_m is the last element in the chain, Mallory can always remove all entries between P_m and any previous entry by a colluding party, e.g., P_e . This DOS attack cannot be prevented through technical means alone.

Theorem 3. *When checksums are constructed using formula from Section 3.1.3, users cannot repudiate an entry (I5).*

Proof. (sketch) This follows by construction when integrity checksum C_i is implemented using the non-repudiable signatures of Section 3.1.3. \square

Theorem 4. *An adversary cannot successfully claim that a valid provenance chain for a given document belongs to a document with different contents (I6).*

Proof. (sketch) This follows directly from the collision-free nature of hashing and the fact that a hash of the current document contents is included in each chain entry, which is then authenticated using the chained C_i checksums. Substituting the chain for a different document will be detected by a *super auditor* when a checksum fails to verify. \square

Theorem 5. *If a document's contents are inconsistent with its history as recorded in a provenance chain with a reversible or plaintext representation for w_i fields, then any superauditor can detect the discrepancy (I7).*

Proof. By definition, a *superauditor* can decrypt all details of the w_i field in each entry, if the w_i fields are encrypted with session keys. Otherwise, the w_i fields are available in plaintext. After verifying the chain, the superauditor can apply the w_i entries in reverse, repeatedly verifying that the hash of D_i included in entry P_i

matches the hash of its recreated contents. At the last verification, D_0 should be the empty document. \square

If a non-reversible representation is used for the w_i entries, or the auditor is not a superauditor, the auditor may still be able to tell that the chain is inconsistent with the current contents of D . E.g., if a chain entry says that all document appendices were deleted, and no subsequent entry added any appendices, then application-domain-dependent reasoning lets an auditor conclude that something is wrong if the document has appendices.

Theorem 6. *Any auditor can verify the chain (C1). Auditors can only decode entry details for which they are authorized.*

Proof. (sketch) This also follows directly by construction. When deploying non-repudiable signature-based checksums as in Section 3.1.3, chain verification involves only public key signature operations and no other secret values. It can thus be performed by any party.

Now consider the question of whether unauthorized parties can access the details of entries. We argue the case where a single session key k_i is used to encrypt all sensitive details in the entry, and the key itself is protected using broadcast encryption; the argument is similar if multiple session keys or a single shared key are used for this purpose. First, a session key k_i for W_i is accessible only to principals that can retrieve it by decrypting at least one item in set I_i . If a principal can decrypt one of these items, then it possesses a private key in the broadcast encryption tree, and must therefore be an auditor represented by a leaf in the subtree rooted at the private key in question. Thus the principal is an auditor who should be allowed to obtain the session key, and therefore should be allowed to see all data encrypted with it, including W_i and (if encrypted) U_i and $public_i$. \square

Finally, we discuss chain verifiability when cryptographic commitments are used for potentially sensitive plaintext, and the plaintext is subsequently removed.

Theorem 7. *The use of cryptographic commitments in place of potentially sensitive plaintext subfields of U_i or W_i does not affect the verifiability of the chain.*

Proof. (sketch) The checksum component of the entry is computed by using cryptographic commitments rather than the sensitive data item's plaintext. Hence, if the plaintext is removed when releasing to an untrusted principal, the chain remains verifiable, as the verification mechanism only requires the commitment. The chain integrity is also not compromised, due to unforgeability of signatures on the checksum entries for other users. \square

4 Empirical Evaluation

Several avenues are available for implementing secure provenance functionality: in the operating system kernel, at the file system layer, or in the application realm.

Kernel Layer. In this implementation approach, provenance record functions are handled by trapping kernel system calls, similar to the approach taken in the Provenance-aware Storage System (PASS) [38]. The main advantage of this approach is its transparency to user level applications and the file system layer. Major drawbacks include the fact that the logic and higher level data management semantics are not naturally propagated to the kernel, thus limiting the types of provenance-related inferences that can be made. Yet another drawback of such an approach is its limited portability, as any new deployment platform will require porting efforts.

File System Layer. The file system can be made provenance-aware and augmented to transparently handle securing collected provenance information. Similarly to the kernel layer implementation, one of the main advantages of such an approach is transparency. However, persisting provenance state transparently inside the file system layer will reduce the portability of the provenance assurances, e.g., when provenance-augmented files traverse non-compliant environments.

Application Layer. In this approach, the provenance mechanisms are offered through user-level libraries. This can still maintain the transparency of the previous approaches while also allowing for a high degree of portability, i.e., by being independent of kernel and file system layer instances. The provenance libraries can be layered on top of any file system, making rapid prototyping and deployment very easy. Moreover, through dynamic linking and by maintaining a compatible interface, existing user applications do *not* need to be recompiled for provenance-awareness.

4.1 The Sprov Library

We implemented a prototype of the secure provenance primitives as an application layer C library, consisting of wrapper functions for the standard file I/O library *stdio.h*. The resulting library is fully compatible with *stdio* functionality, in addition to transparently handling provenance assurances. We used the basic model introduced in Section 3.1.2 and 3.1.3 in this prototype.

In *Sprov*, a *session* is defined as all the operations performed by a user on a file between file open and close. When a file is opened in write or append mode, *Sprov* initiates a new entry in the provenance chain of the file. Information about the user, application, and environment are collected. During write operations, *Sprov* gathers information about the writes, to the file before it is closed. *Sprov* uses a reversible representation of document modifications; as discussed earlier, this allows strong verifi-

cation of the relationship between current document contents and the document's provenance chain. The provenance chain can be used as a rollback log, which can form the basis of a versioning file system; we leave this for future work.

At file close, the session ends. *Sprov* writes a new entry in the provenance chain for the changes made during this session. At this point, the cryptography (implemented using openssl [54]) associated with the chain integrity constructs is executed, as described in Section 3.1. The provenance chain is stored in a separate meta-file for portability.

We provide utilities to facilitate provenance collection and transfer. When a user logs into her system, the *login* utility is invoked, which initializes the session keys and loads the user's preferences and list of trusted auditors. Copying and deletion of a provenance-enabled file uses the *pcopy* and *pdelete* utilities, respectively, as discussed in Section 3.1.6. Finally, the *pmonitor* daemon periodically scans for and removes expired provenance chains.

4.2 Experiments

Our experiments employed x86 Pentium 4 3.4GHz hardware with 2GB of RAM, running Linux (Suse) at kernel version 2.6.11. In this configuration, each 1024-bit DSA signature took 1.5ms to compute. The experiments used a mix of four of the following drive types: Seagate Barracuda 7200.11 SATA 3Gb/s 1TB, 7200 RPM, 105 MB/s sustained data rate, 4.16ms average seek latency and 32 MB cache, and Western Digital Caviar SE16 3 Gb/s, 320GB, 7200 RPM, 122 MB/s sustained data rate, 4.2ms average latency and 16MB cache.

We conducted our experiments using multiple benchmarks in a quest to match several different deployment settings of relevance. In each case, we compared the execution times for the baseline unmodified benchmark (with no provenance collection at all), with a run with secure provenance enabled. We deployed (i) PostMark [26] – a standard benchmark for file system performance evaluation, (ii) the Small and Large file microbenchmark that has been used to evaluate the performance of PASS [38, 48], and (iii) a custom transaction-level benchmark meant to test the performance in live file systems with file sizes distributed realistically [2, 16], and real-life file system workloads [17, 28, 44].

We also evaluated two different configurations for storing the provenance chain. In the first configuration, provenance chains were recorded on disk (**Config-Disk**), while in the second one, provenance chains were stored in a RAM disk, with a *pmonitor* chron daemon periodically flushing the chain to disk (**Config-RD**).

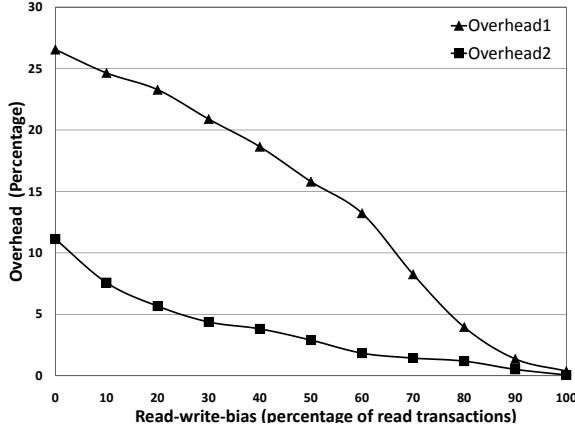


Figure 1: Overhead of secure provenance with Postmark. Overhead1 indicates the overhead using *Config-Disk*, while Overhead2 is from *Config-RD* setting. The overheads are shown from 0% read bias (100% write transactions) to 100% read bias (no write transactions).

4.2.1 Postmark Benchmark

We measured the execution time of the Postmark benchmark [26] with and without the Sprov library. A data set containing 20,000 Postmark-generated binary files with sizes ranging from 8KB to 64KB was subjected to Postmark workloads of 20000 transactions. Each transaction set was a mixture of writes and reads of sizes varying between 8KB and 64KB. We sampled the performance overhead under different write loads by varying the read-write bias from 0% to 100% in 10% increments (i.e. the percentage of write transactions was varied from 100 to 0%). The overheads are illustrated in Figure 1 for both *Config-Disk* and *Config-RD* and range from 0.5% to 11% for *Config-RD*.

4.2.2 Small and Large File Microbenchmarks

The small and large file microbenchmarks [48] have been used in the evaluation of PASS [38]. The small file microbenchmark creates, writes to and then deletes 2500 files of sizes ranging from 4KB, 8KB, 16KB, and 32KB. We benchmarked the overhead for file creation as well as synchronous writes. The results for *Config-Disk* are displayed in Figure 2.

An interesting effect can be observed. Similar to the experiments in PASS, the overhead percentage is quite high for small files and decreases rapidly with increasing file sizes. We believe this effect can be attributed to disk caching. Specifically, for very small file size accesses - which go straight to the disk cache, the main overhead culprit (crypto signatures) dominates. As file sizes increase, additional real disk seeks are incurred in both cases and start to even out the execution times. Eventually, the overhead stabilizes to under 50% for larger

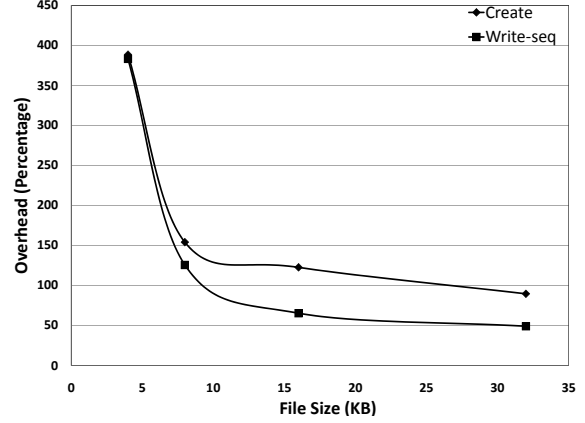


Figure 2: Small file system microbenchmark create and write performance for 2500 files.

files which suggests that roughly 1-3 seek times are paid per file and the secure case adds the equivalent of another seek time (the crypto signature).

	no prov	<i>sprovCD</i>	%Overhead	<i>sprovCRD</i>	%Overhead
Seq-write	13.084	13.328	1.87%	13.308	1.71%
Rand-write	15.211	15.390	1.18%	15.285	0.48%

Table 1: Overhead (in seconds) for large file microbenchmark, under *Config-Disk* (CD) and *Config-RD* (CRD).

The small file microbenchmark only measures the effect of writes to many small files [38]. Often, writes to large files can provide more representative estimates of typical overheads in file systems. Thus, next we deployed the Large file benchmark as described in PASS. We performed the sequential-write and random-write operations of the benchmark. Both unmodified and provenance-enhanced versions of the benchmark were run, and this time, the disk write-caches were turned off to eliminate unwanted disk-specific caching effects.

The benchmark consists of creating a 100 MB file by writing to it sequentially in 256KB chunks, followed by writing 100MB of data in 256KB units written in random locations of the file. The overheads for sequential and random writes are presented in Table 1. In both cases (*Config-Disk* and *Config-RD*), the overheads are considerably lower than the overheads reported in [38], despite the additional costs of recording all the file writes to its provenance chain.

4.2.3 Hybrid Workload Benchmark

Benchmarks like Postmark are useful due to their standardized nature and ability to replicate the results. Additionally we decided to evaluate our overheads in a more realistic scenario, involving practical, documented work-

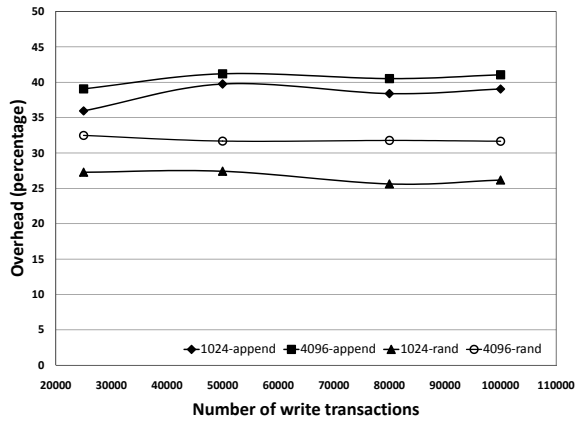


Figure 3: Overhead for different number of write transactions and data sizes, at 100% write load (*Config-Disk*).

loads and file system layouts. We constructed a layout as discussed by Douceur et.al [16], which showed that file sizes can be modeled using a log-normal distribution. We used the parameters $\mu^e = 8.46$, $\sigma^e = 2.4$ to generate a distribution of 20,000 files, with a median file size of 4KB, and mean file size of 80KB, along with a small number of files with sizes exceeding 1GB to account for large data objects, as suggested in [2, 16].

Our first workload on this dataset involved fixed number of write transactions. Under the *Config-Disk* setting, we performed 25K, 50K, 80K, and 100K write transactions. Between each experimental runs, we recreated the dataset, cold-booted the system, and flushed file system buffers to avoid variations caused by OS or disk caching. In each transaction, a file was opened at random, and a fixed amount of data (1KB and 4KB) was written into it. We measured the overhead for both appends and random writes. These are shown in Figure 3. Constant overheads can be observed for each of the 4 configurations, with append situated between 32% and 42%, and random writes between 26% and 33%.

Next, we modeled the percentage of write to read transactions according to the data in [17, 28, 44] which suggest this varies from 1.1% to 82.3%. To this end, we deployed information about workload behavior and used parameters for the instructional (INS), research (RES) [44], a campus home directory (EECS) [17], and CIFS corporate and engineering workloads (CIFS-corp, CIFS-eng) [28]. The RES and INS workloads are read-intensive, with the percentage of write transactions less than 10%. The CIFS workloads are less read-intensive, with the read-write ratio being 2 : 1. The EECS workload has the highest write load, with more than 80% write transactions. The results are shown in Figure 4 and 5, for both the disk-based (*Config-Disk*) and the RAM-disk op-

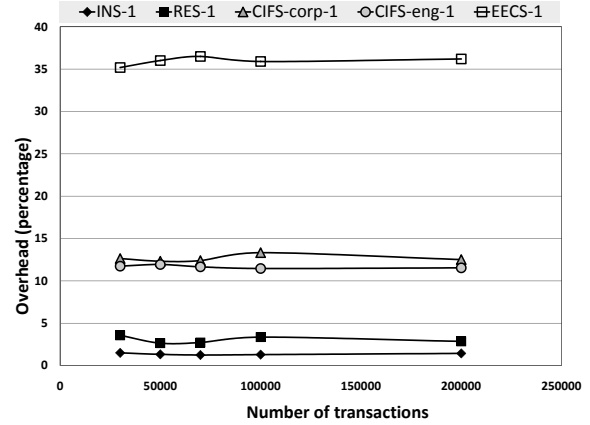


Figure 4: Overhead for various types of workloads (*Config-Disk*).

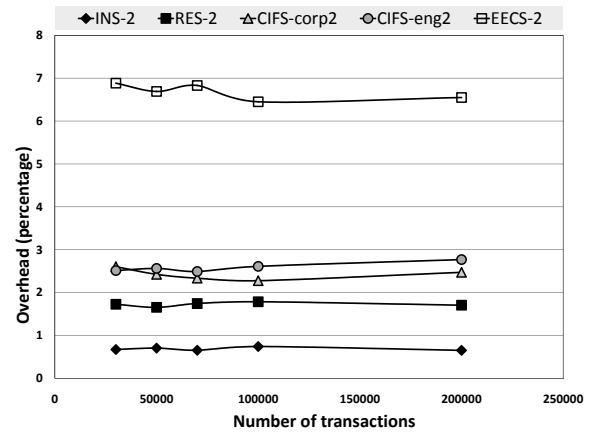


Figure 5: Overhead for various types of workloads (*Config-RD*).

timized (*Config-RD*) modes.

Read-intensive workloads can be seen as almost overhead free, with less than 5% overheads for both RES and INS (this goes down to less than 2% for the RAMdisk-optimized mode). For write intensive workloads, the overheads are higher, but still less than 14% for the CIFS workloads (*Config-Disk*), and less than 36% for EECS (*Config-Disk*). With the RAMdisk-optimization, the overheads go down to less than 3% for CIFS and around 6.5% for EECS.

Summary. Sprov facilitates collection of provenance with integrity and confidentiality assurances, while incurring minimal overhead. Read performance is unaffected by the use of Sprov. Benchmarks show that, with the *Config-RD* setting, use of Sprov incurs an overhead less than 3% in a multitude of realistic workloads.

5 Related Work

Researchers have categorized provenance systems for science [50] and investigated the question of how to capture provenance information, typically through instrumenting workflows and recording their provenance [3, 4, 7, 37, 52, 53]. Other provenance management systems used in scientific computing include Chimera [18] for physics and astronomy, myGrid [60] for biology, CMCS [39] for chemistry, and ESSW [19] for earth science.

Another technique is to collect provenance information at the operating system layer, with the advantage of being hard to circumvent and the disadvantages of being expensive and hard to deploy. The Provenance-aware Storage System (PASS) [8, 38] takes this approach using a modified Linux Kernel. While PASS does not actually record the data written to files, it collects elaborate information flow and workflow descriptions at the OS level. Our techniques of securing provenance chains can be used to augment PASS or any such system to provide the security assurances at minimal cost.

The database community has explored a variety of aspects of provenance, including the notions of why-provenance and where-provenance and how to support provenance in database records and streams (e.g., [9, 10, 11, 12, 57, 59]). Others have examined the applications of provenance to social networks [21] and information retrieval [31].

Overall, the body of research on provenance has focused on the collection, semantic analysis, and dissemination of provenance information, and little has been done to secure that information [8, 25]. One exception is the Lineage File System [46], which automatically collects provenance at the file system level. It supports access control in the sense that a user can set lineage metadata access flags, and the owner of a file can read all of its lineage information. However, this does not meet the challenges (I1-I7,C1-2) for confidentiality, integrity and privacy of provenance information outlined in [8, 25] and discussed in this paper.

Outside the domain of provenance, researchers have used *entanglement* – mechanisms of preserving the historic states of distributed systems in a non-repudiable, tamper-evident manner [32, 45]. This provides similar assurances to the ones sought here for the realm of systems, yet does not handle provenance for information flows and individual data records.

Source code management systems (SCM) target the provenance needs of a particular application domain. For example, Subversion [15], GIT [30], or CVS [5] with secure audit trails can provide integrity assurances for versions in a centralized file system. GIT, Monotone [1], and several other systems also provide support for a distributed infrastructure. These systems employ a logically

centralized model where users maintain local histories and use a virtual (centralized) repository to merge and synchronize their local repositories. Our approach is intended for applications with a more fully decentralized model, where documents and their histories are physically passed between users in separate administrative domains that may not trust one another. In addition, as our approach is intended to meet the needs of many potential applications, we have worked to provide much higher performance than a SCM system requires

Verifiable audit trails for versioning file systems can use keyed hash-chains to protect version history [42]. Under this approach, auditors are fully trusted and share a symmetric key with the file system for creating the MACs. The audit authenticators need to be published to a trusted third party, which must provide them accurately during audits. Our approach must also handle malicious auditors who could easily falsify the audit.

Similarly to audit trails, secure audit logs based on hash chains have been used in computer forensics [47, 51]. Such schemes work under different system and threat models than secure provenance. By their very nature, audit logs are stationary and protect the integrity of local state. In contrast, provenance information is highly mobile and often traverses multiple un-trusted domains. Moreover, audit logs rarely require the selective confidentiality assurances needed for provenance. For example, the mechanisms proposed in [47] secure logs as a whole, but do not allow authentication of individual modifications. Additionally, provenance is usually associated with a digital object (e.g. file). This association introduces attacks that are not applicable to secure audit logs. Finally, a majority of schemes function under the assumption of single (or very few) parties processing the audit log and computing checksums – a different model from the case of provenance chains where multiple principals' access is required throughout the lifetime of a provenance chain.

Secure Untrusted Data Repository (SUNDR)[29] provides a notion of consistency for shared memory (called fork consistency) akin to the integrity property provided for provenance records in our systems. While our techniques for ensuring chain integrity are related to those used in SUNDR, the adversarial model of SUNDR is different from ours. In SUNDR, a set of trusted clients communicate with an untrusted server storing a shared filesystem. In contrast, our system does not employ a central server, and allows any number of users to be corrupted. Moreover, SUNDR does not address confidentiality issues.

Multiply-linked hash chains have been used for signature cost amortization in multicast source authentication [20, 23, 35, 41]. Our spiral chain constructs are similar in principle. One main difference however is that such hash

chains all assume a single sender signing the message block containing the hashes. We can adopt these methods to amortize signature costs in consecutive provenance chain entries from the same principal, but with multiple principals, we need chaining using non-repudiable signatures. Also, many of the hash-chain schemes require the entire stream to be known *a priori*, an assumption not applicable to provenance deployment settings. Finally, the second spiral construction allows integrity-preserving compaction, which is not possible with the hash chains.

Integrity of cooperative XML updates has been discussed in [33], where document originators define a flow path policy before dissemination and recipients can verify whether the document updates happened according to this flow policy. In contrast, for flexibility and wider applicability, our model and integrity assurances do not require the existence of pre-defined flow path policies, in order to provide the integrity assurances described in Section 2.

6 Conclusion

In this paper, we introduced a cross-platform, low-overhead architecture for capturing provenance information at the application layer. Our approach provides fine-grained control over the visibility of provenance information and ensures that no one can add or remove entries in the middle of a provenance chain without detection. We implemented our approach for tracking the provenance of *data writes*, in the form of a library that can be linked with any application. Experimental results show that our approach imposes overheads of only 1–13% on typical real-life workloads.

Acknowledgments

We thank the anonymous reviewers and Alina Oprea for their valuable comments; Bill Bolosky and John Douceur for suggestions about file system distributions; and Kiran-Kumar Muniswamy-Reddy and Margo Seltzer for help with the microbenchmarks. Hasan and Winslett were supported by NSF awards CNS-0716532 and CNS-0803280. Sion was supported by the Xerox Research Foundation and by the NSF through awards CNS-0627554, CNS-0716608, CNS-0708025, and IIS-0803197.

References

- [1] Monotone Distributed Version Control. Online at <http://www.monotone.ca/>, accessed on December 22, 2008.
- [2] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *Proc. of the 5th USENIX conference on File and Storage Technologies (FAST)*, Berkeley, CA, USA, 2007. USENIX Assoc.
- [3] R. Aldeco-Perez and L. Moreau. Provenance-based Auditing of Private Data Use. In *Proc. of the BCS International Academic Research Conference, Visions of Computer Science*, Sept. 2008.
- [4] R. S. Barga and L. A. Digiampietri. Automatic generation of workflow provenance. In Moreau and Foster [36], pages 1–9.
- [5] B. Berliner. CVS II: parallelizing software development. In *Proc. of the Winter 1990 USENIX Conference*, pages 341–352. USENIX Assoc., 1990.
- [6] M. Blum. Coin flipping by telephone. In *Proc. of Crypto*, pages 11–15, 1981.
- [7] U. Braun, S. L. Garfinkel, D. A. Holland, K.-K. Muniswamy-Reddy, and M. I. Seltzer. Issues in automatic provenance collection. In Moreau and Foster [36], pages 171–183.
- [8] U. Braun, A. Shinnar, and M. Seltzer. Securing provenance. In *Proc. of the 3rd USENIX Workshop on Hot Topics in Security (HotSec)*, July 2008.
- [9] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *Proc. of the ACM International Conference on Management of Data (SIGMOD)*, pages 539–550, New York, NY, USA, 2006. ACM Press.
- [10] P. Buneman, A. Chapman, J. Cheney, and S. Vansummeren. A provenance model for manually curated data. In Moreau and Foster [36], pages 162–170.
- [11] P. Buneman, S. Khanna, and W. C. Tan. Data provenance: Some basic issues. In *Proc. of the 20th Conference on Foundations of Software Technology and Theoretical Computer Science (FST TCS)*, pages 87–93, London, UK, 2000. Springer-Verlag.
- [12] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. *Lecture Notes in Computer Science*, 1973:316–330, 2001.
- [13] Centers for Medicare & Medicaid Services. The Health Insurance Portability and Accountability Act of 1996 (HIPAA). Online at <http://www.cms.hhs.gov/hipaa/>, 1996.
- [14] A. Chapman, H. Jagadish, and P. Ramanan. Efficient provenance storage. In *Proc. of the ACM SIGMOD/PODS Conference*, Vancouver, Canada, 2008.
- [15] B. Collins-Sussman. The subversion project: Buiding a better CVS. *Linux J.*, 2002(94):3, 2002.
- [16] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *Proc. of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 59–70. ACM Press, 1999.
- [17] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, pages 203–216, Berkeley, CA, USA, 2003. USENIX Assoc.
- [18] I. T. Foster, J.-S. Vockler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proc. of the 14th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 37–46, Washington, DC, USA, 2002. IEEE Computer Society.
- [19] J. Frew and R. Bose. Earth system science workbench: A data management infrastructure for earth science products. In *Proc. of the 13th International Conference on Scientific and Statistical Database Management (SSDBM)*, page 180, Washington, DC, USA, 2001. IEEE Computer Society.
- [20] R. Gennaro and P. Rohatgi. How to Sign Digital Streams. *Information and Computation*, 165(1):100–116, 2001.
- [21] J. Golbeck. Combining provenance with trust in social networks for semantic web content filtering. In Moreau and Foster [36], pages 101–108.
- [22] O. Goldreich. *Foundations of Cryptography*. Cambridge University Press, 2001.
- [23] P. Golle and N. Modadugu. Authenticating streamed data in the presence of random packet loss. In *Proc. of the Symposium on Network and Distributed Systems Security (NDSS)*, pages 13–22, 2001.

- [24] D. Halevy and A. Shamir. The LSD broadcast encryption scheme. In *Proc. of the 22nd Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 47–60, London, UK, 2002. Springer-Verlag.
- [25] R. Hasan, R. Sion, and M. Winslett. Introducing secure provenance: problems and challenges. In *Proc. of the ACM workshop on Storage security and survivability (StorageSS)*, pages 13–18, New York, NY, USA, 2007. ACM.
- [26] J. Katcher. Postmark: a new file system benchmark. Network Appliance Tech Report TR3022, Oct 1997.
- [27] N. Kogan, Y. Shavitt, and A. Wool. A practical revocation scheme for broadcast encryption using smartcards. *ACM Trans. Inf. Syst. Secur.*, 9(3):325–351, 2006.
- [28] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proc. of the USENIX Annual Technical Conference*, pages 213–226, Berkeley, CA, USA, 2008. USENIX Assoc.
- [29] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–136, 2004.
- [30] J. Loeliger. Collaborating with Git. *Linux Magazine*, June 2006.
- [31] C. A. Lynch. When documents deceive: Trust and provenance as new factors for information retrieval in a tangled web. *Journal of the American Society for Information Science and Technology*, 52(1):12–17, 2001.
- [32] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proc. of the 11th USENIX Security Symposium*, pages 297–312, Berkeley, CA, USA, 2002. USENIX Assoc.
- [33] G. Mella, E. Ferrari, E. Bertino, and Y. Koglin. Controlled and cooperative updates of XML documents in byzantine and failure-prone distributed systems. *ACM Trans. Inf. Syst. Secur.*, 9(4):421–460, 2006.
- [34] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.
- [35] S. K. Miner and J. Staddon. Graph-based authentication of digital streams. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 232–246, 2001.
- [36] L. Moreau and I. T. Foster, editors. *Provenance and Annotation of Data, International Provenance and Annotation Workshop (IPAW)*, volume 4145 of *Lecture Notes in Computer Science*. Springer, 2006.
- [37] L. Moreau, P. Groth, S. Miles, J. Vazquez-Salceda, J. Ibbotson, S. Jiang, S. Munroe, O. Rana, A. Schreiber, V. Tan, and L. Varga. The provenance of electronic data. *Commun. ACM*, 51(4):52–58, 2008.
- [38] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-aware storage systems. In *Proc. of the USENIX Annual Technical Conference*, pages 43–56, 2006.
- [39] J. D. Myers and et al. A collaborative informatics infrastructure for multi-scale science. In *Proc. of the 2nd International Workshop on Challenges of Large Applications in Distributed Environments (CLADE)*, page 24, Washington, DC, USA, 2004. IEEE Computer Society.
- [40] National Assoc. of Insurance Commissioners. Graham-Leach-Bliley Act, 1999. www.naic.org/GLBA.
- [41] A. Perrig, R. Canetti, D. Tygar, and D. Song. Efficient authentication and signing of multicast streams over lossy channels. In *Proc. of the IEEE Symposium on Security & Privacy*, pages 56–73, May 2000.
- [42] Z. N. J. Peterson, R. Burns, G. Ateniese, and S. Bono. Design and implementation of verifiable audit trails for a versioning file system. In *Proc. of the 5th USENIX conference on File and Storage Technologies (FAST '07)*, pages 93–106, Berkeley, CA, USA, 2007. USENIX Assoc.
- [43] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.
- [44] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proc. of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2000. USENIX Assoc.
- [45] D. Sandler and D. S. Wallach. Casting votes in the auditorium. In *Proc. of the USENIX Workshop on Accurate Electronic Voting Technology*, Berkeley, CA, USA, 2007. USENIX Assoc.
- [46] C. Sar and P. Cao. Lineage file system. Online at <http://crypto.stanford.edu/cao/lineage.html>, January 2005.
- [47] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur.*, 2(2):159–176, 1999.
- [48] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: a performance comparison. In *Proc. of the USENIX 1995 Technical Conference*, pages 249–264, Berkeley, CA, USA, 1995. USENIX Assoc.
- [49] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
- [50] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, September 2005.
- [51] R. Snodgrass, S. Yao, and C. Collberg. Tamper detection in audit logs. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 504–515. VLDB Endowment, 2004.
- [52] M. Szomszor and L. Moreau. Recording and reasoning over data provenance in web and grid services. In *Proc. of the International Conference on Ontologies, Databases and Applications of Semantics (ODBASE)*, volume 2888 of *Lecture Notes in Computer Science*, pages 603–620, Catania, Sicily, Italy, Nov. 2003.
- [53] V. Tan, P. Groth, S. Miles, S. Jiang, S. Munroe, S. Tsasakou, and L. Moreau. Security issues in a SOA-based provenance system. In Moreau and Foster [36], pages 203–211.
- [54] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. www.openssl.org, April 2003.
- [55] The U.S. Securities and Exchange Commission. Rule 17a-3&4, 17 CFR Part 240: Electronic Storage of Broker-Dealer Records. Online at http://edocket.access.gpo.gov/cfr_2002/aprqtr/17cfr240.17a-4.htm, 2003.
- [56] U.S. Public Law No. 107-204, 116 Stat. 745. The Public Company Accounting Reform and Investor Protection Act, 2002.
- [57] N. N. Vijayakumar and B. Plale. Towards low overhead provenance tracking in near real-time stream filtering. In Moreau and Foster [36], pages 46–54.
- [58] D. Wichers, D. Cook, R. Olsson, J. Crossley, P. Kerchen, K. Levitt, and R. Lo. PACLS: An access control list approach to anti-viral security. In *Proc. of the 13th National Computer Security Conference*, pages 340–349, 1990.
- [59] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2005.
- [60] J. Zhao, C. A. Goble, R. Stevens, and S. Bechhofer. Semantically linking and browsing provenance logs for E-science. In *Proc. of the 1st International IFIP Conference on Semantics of a Networked World (ICSNW)*, pages 158–176, 2004.

Causality-Based Versioning

Kiran-Kumar Muniswamy-Reddy and David A. Holland
{kiran, dholland}@eecs.harvard.edu
Harvard School of Engineering and Applied Sciences

Abstract

Versioning file systems provide the ability to recover from a variety of failures, including file corruption, virus and worm infestations, and user mistakes. However, using versions to recover from data-corrupting events requires a human to determine precisely which files and versions to restore. We can create more meaningful versions and enhance the value of those versions by capturing the causal connections among files, facilitating selection and recovery of precisely the right versions after data corrupting events.

We determine when to create new versions of files automatically using the causal relationships among files. The literature on versioning file systems usually examines two extremes of possible version-creation algorithms: open-to-close versioning and versioning on every write. We evaluate causal versions of these two algorithms and introduce two additional causality-based algorithms: Cycle-Avoidance and Graph-Finesse.

We show that capturing and maintaining causal relationships imposes less than 7% overhead on a versioning system, providing benefit at low cost. We then show that Cycle-Avoidance provides more meaningful versions of files created during concurrent program execution, with overhead comparable to open/close versioning. Graph-Finesse provides even greater control, frequently at comparable overhead, but sometimes at unacceptable overhead. Versioning on every write is an interesting extreme case, but is far too costly to be useful in practice.

1 Introduction

Versioning file systems automatically create copies (i.e., versions) of files as they are modified, providing numerous benefits to users and administrators. Users find versions convenient when they inadvertently remove or corrupt a valuable file. Administrators find that versioning systems greatly reduce the rate of requests to restore files

from backup. In addition, versioning file systems provide the means to clean up after a data-corrupting intrusion. Unfortunately, versioning alone does not help in identifying the most recent “good” version of a file or how data corruption may have spread from one file to another.

Snapshotting, often implemented using checkpoints, is another approach for versioning that is common for backup systems. Such a system periodically takes a whole or incremental image of the file system and then uses copy-on-write for data modified after the snapshot. Snapshots, similar to versioning file systems, cannot help identify the most recent “good” version of a file. Another drawback of snapshot-based systems is the granularity of recovery: it is not possible to undo changes made between snapshots.

Several new file system designs capture causality relationships among files for a variety of different purposes. For example, Taser [7] captures causality information to address the challenge of identifying files tainted by an intrusion or corrupted by administrative errors. Back-Tracker [11] captures causality information to analyze intrusions. Provenance-aware storage systems (PASS) capture the provenance or digital history of files to let users answer questions such as, “How do these two files differ?” “What files are derived from this one?” “From what files is this file derived?” “How are these two files related?” [14]. Other systems [19] preserve causal relationships to enhance personal search capabilities.

Combining versioning and the capture of causal relationships introduces functionality not available in existing systems. For example, suppose a system has been compromised by a data-corrupting worm. Upon identifying a tainted file, the causal relationships provide a mechanism to trace backwards to find the last version prior to the corruption and then trace forward to identify *all* the files tainted by that corruption. These two traces precisely identify the appropriate files and versions that need to be restored to recover from the intrusion. Without versioning, an administrator’s only recourse is to re-

store the system to a clean snapshot, potentially losing valuable user data. Without causal relationships, the administrator cannot know how the corruption has spread.

Similarly, imagine a scenario where a physics simulator produces results today that differ from those produced yesterday. Here, causal data can reveal the cause of the difference, while versioning data can recover to the earlier (and presumably correct) version.

Conventional versioning file systems [3, 10, 15, 18, 24] typically use one of two techniques to determine when to create new file versions: “open-close” and “version-on-every-write”. In the “open-close” approach, versions are defined relative to `open` and `close` events. Typically a new version is created upon the first block update after an `open` and all writes that occur before the final `close` operation appear in that new version. This has the potential to lose valuable information. For example, consider the split-logfile vulnerability in Apache 1.3 [25]. The vulnerability, present in a helper program called `split-logfile`, allows any file in the system with a `.log` file extension to be written. Assume that a database server running on the same machine as the Apache `split-logfile` helper uses a file called `db.log` to store its recovery information. This database server opens the file when it is started and keeps it open. The first time the database server writes to the log file, an “open-close” system will create a new version of it. That version will remain the current version until the database is shut down. Now, suppose an attacker exploits Apache and writes new data in `db.log`. At this point, the log consists of some old “good” log entries and some new “bad” log entries. Even if an administrator finds that `db.log` has been corrupted, the only version available for recovery is the one that existed before the database server wrote anything. If the administrator restores that version, all database operations since the server started will be lost. One might turn to the “version-on-every-write” algorithm, which creates a new version each time data is written to the file; this approach ensures that no data is lost, but it can be expensive in both time and space.

Fortunately, versioning algorithms informed by causality relationships produce versions that facilitate recovery to the pre-tampering state without the overhead of versioning on every write. In the Apache example above, causality-based techniques force a new version of the log file to be started before the attacker’s writes are applied. We introduce two such causality techniques: **Cycle-Avoidance** and **Graph-Finesse**. Cycle-Avoidance conservatively declares new versions using knowledge local to the objects being acted upon (i.e, process, files, pipes, etc). **Graph-Finesse** is less conservative, using global knowledge to maintain an in-memory graph of dependencies between objects, declaring a new version when-

ever adding a dependency edge introduces a cycle. We discuss these algorithms in more detail in Section 4.

As we show in Section 6, any kind of versioning, when coupled with maintenance of causal relationships, provides significant value. In the presence of long-running and/or concurrent execution, Cycle-Avoidance creates the versions necessary to recover from corruption without introducing significant overhead above that of open/close. Graph-Finesse creates slightly fewer versions than Cycle-Avoidance, but it pays significant overhead in workloads that read and write a large number of files. Versioning on every write exhibits sufficiently high overhead that it is impractical.

The contributions of the paper are as follows:

- New functionality arising from the integration of versioning with causal data,
- New causality-based techniques for versioning,
- A prototype embodying versioning and causal relationships, and
- An evaluation of four causality-based versioning algorithms.

The rest of the paper is organized as follows. In Section 2, we introduce the system upon which we build our causality-based versioning system and describe its essential architectural details. In Section 3, we discuss several novel use cases that causality-based versioning enables. In Section 4, we present details of the new versioning algorithms. In Section 5, we discuss the versioning file system implementation. In Section 6, we present evaluation results. In Section 7, we discuss related work. Finally, we conclude in Section 8.

2 Causal Relationships with PASS

We extended PASS [14] (Provenance-aware storage system), the causality-collection system we built, to capture versioning information. We chose PASS as it captured precisely the data that we needed and has a modular architecture that made it easy to add versioning. In this section, we provide a high level overview of the PASS architecture to provide the necessary background to understand our version creation algorithms and implementation.

Figure 1 shows the PASS architecture. Its key components are:

- **Interceptor:** The interceptor intercepts system calls, passing information to the observer, described next. The interceptor is a thin layer that is operating system specific, while the remaining components can be operating system independent.

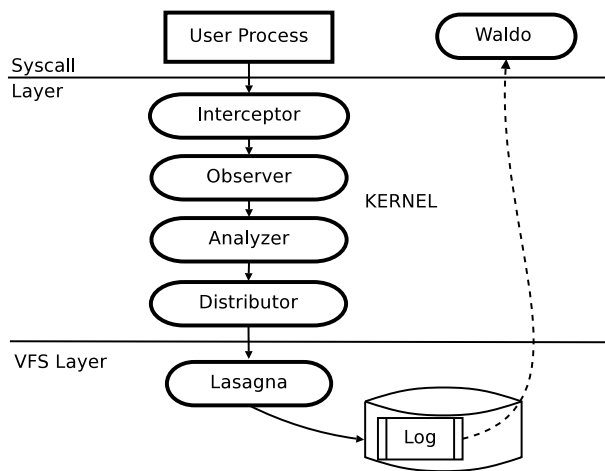


Figure 1: PASS Architecture

- **Observer:** The observer translates system call events to provenance records. For example, when a process P reads a file A , the observer generates a record $P \rightarrow A$, to indicate that the process P depends on the file A . It is precisely these provenance events that capture the causal dependencies in which we are interested.
- **Analyzer:** The analyzer processes the stream of provenance records, making sure that there are no cyclic dependencies among objects. This is where we implement our different versioning algorithms.
- **Distributor:** The distributor maintains provenance for transient objects such as pipes and processes. When these transient objects become part of the ancestry of a regular file on a PASS volume, the distributor creates a virtual object for them on the PASS volume and stores their records to the volume. Creating a virtual object avoids the need for duplicating the provenance of transient objects each time we need to create a causal dependency involving them. Similar to regular files, PASS versions these transient objects when necessary.
- **Lasagna:** Lasagna is the provenance-aware file system that stores provenance records along with the data. Internally, lasagna writes the provenance into a log.
- **Waldo:** Waldo is a user-level daemon that reads provenance records from the log and stores them in a database. After a data corruption, PASS's recovery tools consult this database to determine causal relationships.

The observer and analyzer are central to causality-based versioning and are discussed further in Section 4.

PASS maintains provenance, and therefore causal relationships, for both persistent and transient objects. A relationship between two files (persistent objects) is therefore expressed indirectly as two relationships, each between a file and a process. For example, when a process issues a `read` system call, PASS creates a record stating that the process causally depends upon the file being read. When that process then issues a `write` system call, PASS creates a record stating that the written file depends upon the process that wrote it. The causal relationships described by these records are the heart of the data we use, both to instantiate file versions and also to choose files to restore after a corruption.

3 Use Cases

In this section, we discuss use cases that demonstrate the novel functionality enabled by causality-based versioning.

3.1 Intrusion Recovery

Recently, one of the authors upgraded the software on his system. The upgraded packages included `coreutils`, the package that contains `ls`. This author completed the upgrade and continued working for the rest of the evening. However, when he came back the next morning, `ls` emitted the message:

```
/bin/ls: unrecognized prefix: do
/bin/ls: unparsable value for
      LS_COLORS environment
      variable
```

The author then searched the web to learn that the behavior might be the result of an intrusion. He promptly installed `chkrootkit` and `rkhunter`, two popular programs to verify that a system has been hacked. However, both the programs failed to locate any known rootkits. At this point, it was unclear if the aberrant behavior of `ls` was due to the update to `coreutils` the evening before or an intrusion.

Had he been running PASS, the author could have followed the chain of provenance dependencies of the file `/etc/DIR_COLORS`. (`LS_COLORS` is derived from `DIR_COLORS`). If the provenance chain of `DIR_COLORS` indicated that it was modified by the package manager or a legitimate system utility, then the system had not been hacked; otherwise, the system had been hacked. In the event the system had been hacked, there would be only one option: *wipe the system and re-install*. This is obviously undesirable, especially since the author had recently completed a re-install in order to

upgrade. Faced with this situation, the author longed for a versioning file system coupled with PASS that would permit him to selectively roll back the affected files to the version just before they were corrupted. Had that been an option, he could have continued using his system without a full re-install.

3.2 Reproducing Research Results

Systems that collect provenance frequently do so to facilitate the reproduction of scientific results [22]. Consider the common scenario where a scientist collects data from some device (e.g., a telescope), transforms it through many intermediate stages and produces a final output file. Suppose he finds, a few months after publishing the data, that one of the programs in the intermediate stages had a bug. The scientist has no option but to begin anew with the raw data and then re-run all the experiments that he thinks may have been affected by the bug.

If, however, he has complete provenance of his data, he can identify precisely which data sets were affected by the corrupt program. Hence, he need only re-run those experiments from the point at which the corruption occurred. Further, if the raw data is unavailable (frequently, raw data is archived and removed from data processing systems once it has undergone its initial pre-processing), he can use a versioning system to recover the missing raw data to re-run the experiments. This method obviates the need to retrieve the raw data from archives or a central repository.

3.3 System Configuration Management

Software configuration management is extremely hard [26]. New software installation can (and regularly does) break existing software, because packages interact with each other through various agents: libraries, registries, configuration files, and even environment variables.

Provenance systems can help alleviate some of the problems of configuration management by helping users recover from a corrupt configuration. One of our authors recently installed a new music player on his system. The music player, in turn, depended on a number of libraries that needed to be updated or downloaded. After the install, the music player worked well, but much to the annoyance of the author, his movie player ceased to work. The author guessed that it was probably because of the updates to the libraries. The author tried to use the package manager to revert the system to the state that existed prior to the music player install. Using the system package manager to remove the music player did not help, because as far as the package manager was concerned, the library updates were independent of the mu-

sic player install. The author could not manually undo the library updates as he did not know the list of libraries that were installed. A record of the causal relationships between the libraries, the music player, and the movie player would have helped the author identify which of the libraries were common to the music player and the movie player and hence would have helped to point out (or narrow down) the offending library. Such causal data coupled with a versioning file system provides exactly the information needed to permit the user to revert all the modified files to a state prior to that of the music player install. Since the versioning system even restores the package manager database to its prior state, it preserves the consistency of the system.

The problems outlined in this use case arise mainly because package management dependencies are generated manually and are brittle in nature. Alternatively, one could use causal data recorded by PASS to gather the true dependencies of a package that, in turn, can help perform better roll backs after installation.

3.4 Database Recovery

Traditional databases are designed to recover from software and hardware crashes. However, those mechanisms are not sufficient to recover from a human error or a compromise due to the time gap between the event occurrence and the detection. In such cases, recovery involves a manual sanitizing of the database. Causality-based versioning can help reduce the amount of effort and the downtime of the service as we show in the following example. For simplicity, we assume that the database is not running transactionally.

A faulty client can corrupt a database by either adding incorrect entries, removing valid entries, or updating existing entries incorrectly. Once the faulty client is detected, one can use the causal data collected by PASS with the versioning data to recover the database. Recovery is simple in the case where the last database update is by the faulty client. In this case, recovery simply entails reverting the database to a version before the client updated it. In the case where legitimate actions are interleaved with the actions of the faulty client, automatic recovery is hard as both legitimate and faulty updates are coalesced in main memory and then written to disk. In this scenario, one can use causal information to recover the database to a version before the faulty client's modifications and a version after the faulty client's modifications and then compute a difference of the data dump between the two versions. The difference in the data dump will contain both legitimate and illegitimate data that needs to be sanitized, but the number of rows requiring manual checking is much smaller than the entire database.

3.5 Intellectual Property Compliance

Causality-based versioning can also help verify intellectual property (IP) compliance and enable removal of IP violations. For example, companies that use and develop both proprietary software and open source software routinely require pre-release checks to make sure the proprietary software has not been tainted by open source software and vice-versa. In most cases, this is a tedious, manual process. One can use causal relationships to identify paths between source files with different licensing models. When coupled with a versioning file system, the system supports rigorous analysis of such license pollution and potentially explicit means of reverting to untainted states.

4 Versioning Algorithms

As described in Section 2, in the PASS architecture the *observer* generates causal relationship data and the *analyzer* prunes these relationships, removing duplicates and cycles. Programs generally perform I/O in relatively small blocks (e.g., 4 KB), issuing multiple reads and writes when manipulating large files. Each `read` or `write` call causes the observer to emit a new record, most of which are identical. The analyzer removes these duplicates. Meanwhile, cycles can occur when multiple processes are concurrently reading and writing the same files [2]. Cycles in causality are nonsensical and must be avoided. In the PASS system, the analyzer prevents cycles by forcing new versions of objects to be created. It does this by choosing when to *freeze* them; that is, when to declare the current version “finished” and begin a new version. Transient objects (processes and pipes) can also be frozen to break cycles. To experiment with causality-based versioning, we installed the versioning algorithms in the analyzer. In this section, we describe the versioning algorithms we used, referencing Table 1 as an example sequence of events.

4.1 Traditional Algorithms

In the open-close (OC) algorithm, the last close of a file (that is, when no more processes have the file open) triggers a freeze operation. The next open and write triggers the start of a new version. This algorithm does not preserve causality; some sequences of events (including the example in Table 1) produce cycles. Figure 2 illustrates how this happens.

The version-on-every-write (ALL) algorithm creates a new version on every write. This avoids any violations of causality but potentially creates a large number of versions. In this sense, it is the most conservative of the algorithms we consider. The code for this algorithm is

Step	P	Q
1	read A	
2		read B
3	write B	
4		write A
5	read A	
6		read B

Table 1: Example scenario to illustrate the versioning algorithms. Each `read` and `write` operation is enclosed by an `open` and `close`. All objects are initially at version one.

quite simple; because each write results in a new version of a file and each read results in a new version of a process, each record refers to a distinct version of something. Thus, there is no need to check for either duplicates or cycles.

4.2 Cycle-Avoidance

The Cycle-Avoidance (CA) algorithm, as its name suggests, preserves causality by avoiding cycles. For each object, the analyzer maintains a unique object ID (assigned at object creation), a version number (incremented on each freeze), and an *ancestor table*. The ancestor table records the object ID and version number of all the immediate ancestors of the object. When CA receives a record of the form $A_i \rightarrow B_j$, it stores B_j in the ancestor table of A . CA creates a new version of an object whenever it adds a *new* ancestor, where different versions are considered distinct, to the object’s ancestor table. Doing so guarantees that no cycles will be created. CA differs from version-on-every-write, because not all writes add new ancestors.

When the analyzer receives a record of the form $A_i \rightarrow B_j$, it examines the ancestor table of A for B_k , that is, some version k of object B , and uses the following rules to perform both duplicate detection and cycle handling.

- **Rule CA.1:** If no B_k exists in the ancestor table of A , then B is a new ancestor for A . Issue a freeze operation on A to create a new version and add B_j to the ancestor table of A .
- **Rule CA.2:** If B_k exists and $j = k$, then the causality record $A_i \rightarrow B_j$ is a duplicate and we discard the record.
- **Rule CA.3:** If B_k exists and $j < k$, the new record refers to a version older than the most recent one recorded in A ’s ancestor table, and the existing causality relationship $A_i \rightarrow B_k$ subsumes any causal relationships of B_j . Hence, the causality

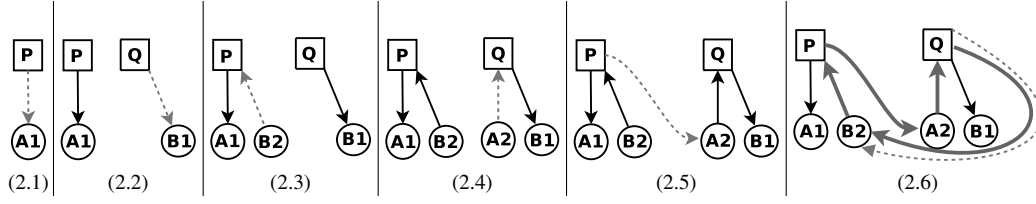


Figure 2: Illustration of the open-close algorithm for the sequence in Table 1. The arrows represent causality and point opposite to data flow. In (2.3), a new version of B is created as it is the first write since the last close. In (2.4), a new version of A is created for the same reason. A cycle $A_2 \rightarrow Q \rightarrow B_2 \rightarrow P \rightarrow A_2$ (thick lines) results on the last read as shown in (2.6).

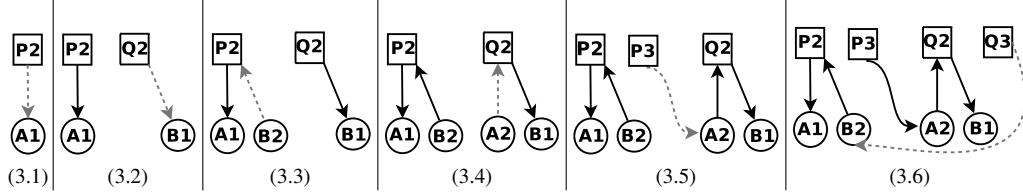


Figure 3: Illustration of the Cycle-Avoidance algorithm for the sequence in Table 1. In (3.1), a new version of P is created by the rule CA.1. In (3.2), a new version of Q is created by the rule CA.1. In (3.3), a new version of B is created by the rule CA.1. In (3.4), a new version of A is created by the rule CA.1. In (3.5), a new version of P is created by the rule CA.4. In (3.6), a new version of Q is created by the rule CA.4. The end result is that there are no cycles.

record $A_i \rightarrow B_j$ is a duplicate and we discard the record.

- **Rule CA.4:** If B_k exists and $j > k$, B_j is a newer version than the B_k in A 's ancestor table. Thus, B_j depends on some objects on which B_k did not depend. Therefore, we issue a freeze on A to create a new version and update the ancestor table of A to name B_j instead of B_k .

Figure 3 illustrates the behaviour of the Cycle-Avoidance algorithm for the example sequence in Table 1.

4.2.1 Self-Cycles

Self-cycles arise when a process is both reading and writing the same file. Some programs, such as the GNU linker, generate self-cycles as they repeatedly read from and write to their output files. The Cycle-Avoidance algorithm as described can create a large number of unnecessary versions in this situation. To avoid this, we track each object's *last ancestor*. When the analyzer receives a record of the form $A_i \rightarrow B_j$, it makes the following check:

- **Rule CA.self-cycle:** If the last ancestor of B_j is A_i , the new record creates a self-cycle; discard the record.

The above rule tells us that the last version change of B occurred *because* of the current version of A . In that

case, the data being fed to A originated from A itself, and we have a self-cycle. Records representing self-cycles do not add information and can be dropped immediately.

4.3 Graph-Finesse

As described above, Cycle-Avoidance decides when to create new versions using local knowledge about the object to which a dependency refers. In contrast, Graph-Finesse (GF) uses global knowledge to make its decisions. It maintains a global directed graph of the causal dependency relationships between objects. The GF algorithm checks each new record against the graph and forces the creation of a new version of a single file if and only if adding the record would otherwise create a cycle. The name arises from the fact that it picks out a comparatively small number of new versions to create while still preserving causality.

Given a record $A_i \rightarrow B_j$, GF uses the following rules:

- **Rule GF.dup:** Check if $A_i \rightarrow B_j$ already exists in the causal-dependency graph. If so, the record is a duplicate; discard it.
- **Rule GF.detect:** Check if $B_j \rightarrow^* A_i$, that is, if a path of zero or more steps exists linking B_j to A_i . If so, then $A_i \rightarrow B_j \rightarrow^* A_i$ forms a cycle. Freeze A , creating A_{i+1} , change the record to $A_{i+1} \rightarrow B_j$, and add this information to the graph. There will now be no cycle; because A_{i+1} is new, it cannot be an ancestor of B_j .

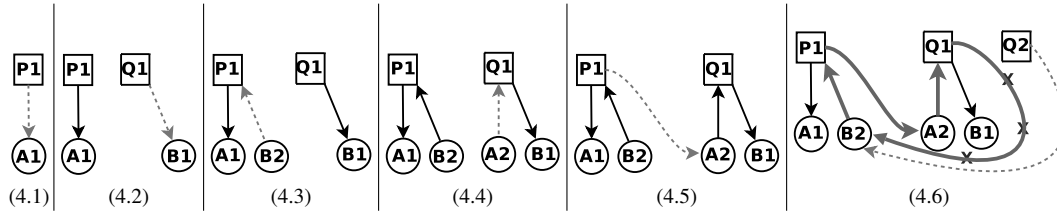


Figure 4: Illustration of the Graph-Finesse algorithm for the sequence in Table 1. In (4.3) and (4.4), new versions of B and A are created by the open-close mechanism. In (4.6), a path exists from B_2 to Q_1 (thick lines), so Q_2 is created by rule GF.detect and no cycle is formed. As we can see, Graph-Finesse creates fewer versions than Cycle-Avoidance.

- **Rule GF.default:** Otherwise, add $A_i \rightarrow B_j$ to the graph.

By design, GF also subsumes open-close versioning and includes the freeze-on-last-close behavior as described in Section 4.1.

To keep the graph from growing without bound, it is important to *prune* it. Any node in the graph (representing some version of some object) may be pruned if that node is *frozen*, that is, any future write to the object will create a new version or be to some already existing newer version, and all the ancestors of the node are frozen. (If an ancestor is unfrozen, writing to it may cause a cycle.)

It is therefore possible to bound the size (or diameter or other measure) of the graph at the cost of creating extra versions, by freezing unfrozen objects that would otherwise be pruned. We have considered this but not implemented it, because there seems to be little need for it in practice.

Graph-Finesse can consume more CPU for some workloads as it has to traverse the graph on every record addition. Our evaluation confirms this. The memory consumption of Graph-Finesse, however, is comparable to the other algorithms. Graph-Finesse can also use the self-cycle logic described in Section 4.2.1. Figure 4 illustrates the behaviour of the Graph-Finesse algorithm for the example workload.

4.4 Discussion

We now discuss how pruning entries in CA and GF does not affect the use cases discussed in section 3. The OC, CA, and GF algorithms record the same versioning data for the first three use cases as they involve an application making a one time modification to all the data files involved. Such changes generate the same causal information and data versions for the three algorithms. The ALL algorithm generates the same causal information and versioning data as the other algorithms with the difference being that the data is spread over more versions. For the remaining two use cases, the causal algorithms record different causal information and versioning data. We ex-

plain how they differ in the database recovery (3.4) use case.

The database server (server) opens the database at startup, writes to it in response to client requests, and closes the file at shutdown. In OC, the first time the server writes to the database, it creates a new version. All subsequent database modifications are part of this new version and old data is not copied before applying these modifications, because the file is still open. Thus, restoring the old version loses any legitimate modifications between the first write and the faulty writes. Clearly, OC versioning is not sufficient for the database recovery use case. ALL has sufficient information as it recorded all causal information and versioned on every database modification. However, versioning the database this frequently is potentially very inefficient.

In CA, reception of a client request produces a new version of the server process. This, in turn, triggers a new version of the database, when the server modifies the database. Multiple modifications resulting from a single client request do not create multiple versions, because the server's version does not change. However, when a new (faulty) request arrives, the server's version increments as does the database's version. Interleaved requests from multiple clients will generate many versions. Thus, in a single client case, CA behaves like OC and when clients interleave, it behaves like ALL. Hence, there is sufficient information to undo a faulty client's updates. The GF algorithm behaves in a manner similar to CA.

5 Implementation

In this section, we describe our versioning design choices, our implementation, and the limitations of our system. Our versioning design was influenced by the comprehensive versioning file system (CVFS) [24], which explored metadata efficiency in versioning file systems. CVFS showed that logging all the modifications of a file to a journal is more efficient than creating a new inode for each version. Hence, we use a redo log for storing versioning data for files. Although CVFS

uses multiversion B-trees to handle versions of directories, we store directory metadata in an undo log, as this is much simpler to implement. We implement our versioning system by modifying Lasagna, the PASS storage engine. Lasagna is a stackable file system, which used eCryptfs [8] as its starting code base.

5.1 File Versioning

For each file `foo`, we maintain a version file `v;12345`, where `12345` is the inode number of `foo`. The version file is a log where we record old data before updating the primary file. Inode numbers are never reused, because we never delete files. For locality, we keep the version files and the files they describe in the same directory. Users cannot access the version file directly as we filter out the version file names in `readdir` and `lookup`.

The version file consists of the following three types of log records. The *version* record marks the start of data records for a version. It contains the version number and the metadata attributes of the version such as the file size, uid, gid, etc. The *page* record holds old data being overwritten. This contains the data and the page number in the file from which the data came. Finally, the *beginptr* record is the last record of a version; it records the location of the corresponding *version* record to allow scanning backwards.

Each version begins with a *version* record, ends with a *beginptr*, and has some number of intervening *page* records. We write a *beginptr* record to the version file when a freeze request is issued on the file. We write a *version* record on the first `write` call on the file after a freeze.

When a program issues a `write` call on a file, we read the pages that the `write` call overwrites and write a *page* record for each. When a file is truncated, we log all discarded pages. We record each page only once per version. For example, if the file system receives two 4K writes at offset 0, we log data only for the first (assuming the version does not change between the writes). On an `unlink`, we rename the target file to `v;12345;deleted` where `12345` is the inode number. A native file system could remove the file blocks from the primary file and append them to the version file. Lasagna, however, is a stackable file system and does not control the file layout of the underlying file system. Instead, we `rename` the file on the last `unlink`. This is more efficient than copying blocks from the primary file to the version file, especially for large files.

5.2 Directory Versioning

For each directory, we maintain (within the directory) a version file named by inode number, as we do for files.

The directory version file has *version* and *beginptr* log records as we do for a regular file. It also has three other log record types: The *add entry* record represents an addition to a directory via `create`, `link`, `mkdir`, or `symlink`. This record contains the inode and version of the directory to which we are adding, and the name, inode, and version of the entry, which can be a file or a directory. The *del entry* record represents a removal from a directory by `unlink` or `rmdir`. This contains the same data as the *add entry* record. The *rename entry* records a directory entry being moved from one directory to another. Where appropriate, this is written to the version logs of both directories involved in the rename. This record contains the inode and version of the new directory, the old directory, the old file, and the overwritten target file (if it existed), and the old and new file names.

Because version logs are never deleted and files are renamed on deletion, directories are never truly empty, so our versioning file system cannot perform `rmdir` operations. Instead, when a directory is removed, we check to see if all the files in the directory are either `v;inode` or `v;inode;deleted` files, i.e., all files are either version files or deleted files. If so, the directory is “virtually” empty, and we can move the directory out of the way using a `;deleted` suffix.

5.3 Accessing Previous Versions

We provide an `ioctl` that is used to access old versions of a file. The `ioctl` takes as input, a name, a version, and a file descriptor and recovers the old version into the file descriptor. Internally, we perform recovery by scanning backward in the version file until we find the desired *version* record. Once this has been found, we scan forward in the version file writing the data pages of the file version to the user supplied file descriptor. We also update the attributes of the file descriptor based on the values recorded in the *version* record. Hence, previous version access is a *redo* operation. Directory operations, on the other hand, are undone depending on the contents of the version log records.

5.4 Limitations

The causal data that we capture is an approximation and can lead to false positives or negatives while performing analysis for recovery. For example, `/etc/shadow` will be in the history of every process and file, as `login` reads it while authenticating users. Hence even legitimate users that log in after an attack can appear to be causally related to the attack. The general approach to deal with this has been to white-list some of these files, i.e., ignore the causal information on some files while performing analysis. Further, contextual policies to ig-

nore some of the causal information can help improve the results of the analysis. For example, to construct the list of files needed to migrate an application, we need to use all the causal information as we do not want the application to fail on restart. However, while analyzing causal data during intrusion recovery, we need consider only those files that have been written by illegal processes.

As with all provenance systems, PASS cannot capture causal dependencies external to the computer. For example, when a user prints a file and then makes some notes based on what she read, PASS cannot capture the dependency between the notes to the source file. PASS does, however, allow users and applications to annotate user-knowledge or application specific provenance to the provenance collected by PASS (with the obvious limitation that the user or application needs to take the correct action).

Attackers can perform a denial of service attack by repeatedly overwriting files, filling the disk with versioning information. While this is not different from an attacker filling the disk with regular files, we can do better than regular file systems by using the causal information to detect anomalous behaviour and prevent it [12, 23].

Because our implementation is a stackable file system, it is vulnerable to tampering by means of unmounting it and inspecting or altering the underlying state. This could be prevented by using cryptography or by having a non-stackable implementation and using a *securelevel*-type scheme to protect raw disk devices.

Finally, we are vulnerable to an intruder changing the kernel. Once the intruder has access to the kernel, she can change the causal information and the versioning information, thus making accurate recovery impossible. Secure Disk Systems [27], where an intruder cannot modify the causal or versioning data once it has been written to disk, helps solve the problem partially, by allowing users to recover data up to the point the system was subverted. This is better than a clean system install. Causal versioning is still useful for recovery in the cases where attackers do not care to cover up their tracks, such as when they set up a bot on a machine and abandon the system after a few days.

6 Evaluation

The goal of our evaluation is twofold. First, to quantify the overheads introduced by the different versioning systems. Second, to evaluate the efficacy and performance of the different algorithms during recovery of files. We address these goals as follows: First, we discuss the evaluation platform and the configurations we used for evaluation. In Section 6.1, we discuss the performance overheads for four benchmarks representative

of a broad range of workloads. In Section 6.2, we discuss how the versioning algorithms perform during recovery.

We ran all the benchmarks on a 3GHz Pentium 4 machine with 512MB of RAM. The machine had a 80GB 7200 RPM Western Digital Caviar WD800JB hard drive that was used to store all file system data and metadata, including causality. The machine was running Fedora Core 5 with a PASS kernel based on Linux 2.6.23.17 and Lasagna was stacked on Ext2. We recorded elapsed, system, and user times, and the amount of disk space utilized for all tests. We also recorded the wait times for all tests; wait time is mostly I/O time, but other factors such as scheduling time can also affect it. We compute wait time as the difference between the elapsed time and system+user times. We do not discuss the user time as it is not affected by the modifications in the kernel. We also ran the same benchmarks on Ext2 using those results as a baseline. In order to separate the overhead due to versioning from the overhead due to causality, we also ran all experiments using versioning without enabling causality collection. We used the open-close algorithm for the latter experiments. We ran each experiment at least 5 times. In all cases, the standard deviations were less than 5%.

We evaluate the system under the following configurations:

VER: open-close versioning with no causal data

OC: open-close versioning with causal data

CA: Cycle-Avoidance versioning with causal data

GF: Graph-Finesse versioning with causal data

ALL: Version-on-every write with causal data

6.1 Performance Overhead Results

We ran the following four workloads to evaluate the versioning algorithms. 1. Linux compile, in which we unpack and build Linux kernel version 2.6.19.1. This benchmark represents a CPU-intensive workload. 2. Postmark, that simulates the operation of an email server. This represents an I/O-intensive workload. We ran 1,500 transactions with file sizes ranging from 4 KB to 1 MB, with 10 subdirectories and 1500 files. 3. Mercurial activity benchmark, where we start with a vanilla Linux 2.6.19.1 kernel and apply, as patches, each of the changes that we committed to our own Mercurial-managed source tree. This benchmark evaluates the overhead a user experiences in a normal development scenario, where the user works on a small subset of the files over a period of time; 4. A biological *blast* [1] workload that is representative of a scientific workload. The workload finds protein sequences that are closely related in two different species. The workload formats two input data files with a tool called *formatdb*, processes the two files with *blast*, and then massages the output data with a

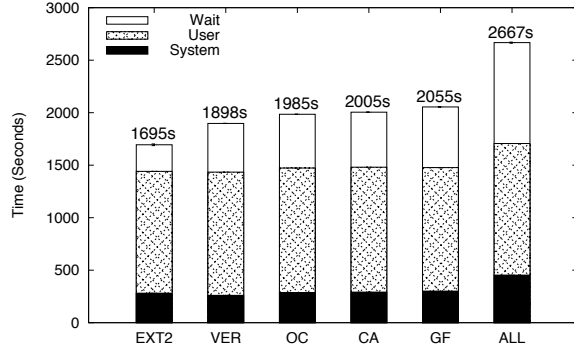


Figure 5: Linux compile elapsed time results.

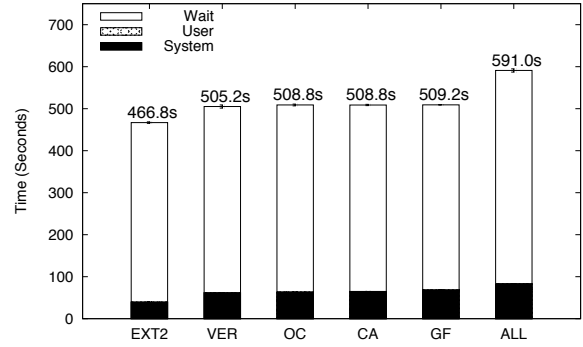


Figure 6: Postmark elapsed time results.

series of *perl* scripts.

	Causal Data	Version Space
VER	-	37.6MB (2.9%)
OC	154.5MB (12.0%)	49.0MB (3.8%)
CA	172.3MB (13.4%)	54.8MB (4.3%)
GF	154.5MB (12.0%)	49.0MB (3.8%)
ALL	462.6MB (35.9%)	1.1GB (85.7%)

Table 2: Linux compile Space overheads. All the overheads shown are computed as a percentage of the data in vanilla Ext2 (1.26GB).

Linux Compile Benchmark Results Figure 5 shows the elapsed time results for Linux compile and Table 2 shows the space overhead. Plain versioning (VER) adds 11.9% to the elapsed time and 2.9% to the space. The increase in elapsed time is mostly due to the additional writes performed to store versions, but a small portion is due to the fact that we use a stackable file system. For the OC, CA, and GF algorithms, the overheads increase moderately over VER to 17.1%, 18.3% and 21.3% respectively. This increase is due to the extra writes issued to record causal data. For this benchmark, CA and GF, the causality based algorithms, perform comparably to OC in terms of elapsed time and version space. ALL, as we expect, has the worst elapsed time performance with 57.4% overhead. The ALL overhead is a result of the enormous number of versions being created and the quantity of data necessary to do so. The system time also increases significantly for ALL due to the distributor having to cache large amounts of causal data.

Postmark Benchmark Results Figure 6 shows the elapsed time results for Postmark and Table 3 shows the space overheads. The overheads follow a pattern similar to the overheads for the Linux compile benchmark. VER

	Causal Data	Version Space
VER	-	1.28GB
OC	1.8MB (0.14%)	1.28GB
CA	1.2MB (0.09%)	1.28GB
GF	1.9MB (0.15%)	1.28GB
ALL	61.2MB (4.74%)	1.38GB

Table 3: Postmark space overheads. Causal data overheads are computed as a percentage of the data written in Ext2 (1.26GB). Postmark deletes all files it creates at the end of the benchmark. In versioning systems, however, no file is deleted and all unlinked files are retained as is. The version space column shows the amount of space retained at the end of each algorithm.

has the lowest overhead at 8.2%. VER’s overhead is due to the extra writes to record version data and the double buffering in Lasagna (stackable file systems cache both their data pages and lower file system data pages). The overheads increase marginally for OC, CA, and GF to 9%, 9%, and 9.1% respectively. The increase is marginal as causal information recorded is minimal and the version data also increases minimally from VER to OC, CA, and GF. Once again, ALL, with a 26.6% overhead exhibits the greatest overhead as expected.

Mercurial Activity Results Figure 7 shows the elapsed time results for the Mercurial activity benchmark and Table 4 shows the space overhead. The performance overheads follow the pattern we have seen so far. However, surprisingly, GF performs worse than even ALL for this benchmark. VER, CA, GF, and ALL have overheads of 25.9%, 28.8%, 27.9%, 89.6%, and 61.3% respectively. The performance overheads of GF is a result of a very large patch combined with the way the program `patch` functions. `patch` works by first reading the patch file and the file to patch, then merges the two files into a temporary file, and finally renames the tem-

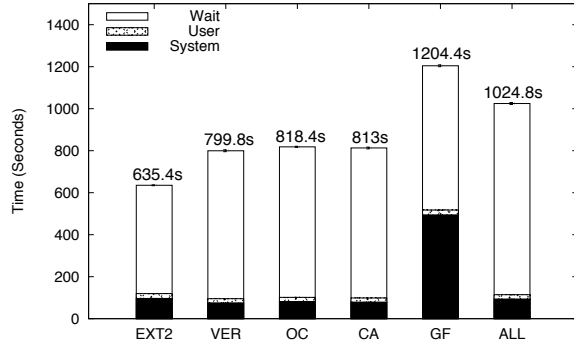


Figure 7: Mercurial activity elapsed time results.

	Causal Data	Version Space
VER	-	228.1MB (26.6%)
OC	38.3MB (4.5%)	233.4MB (27.2%)
CA	28.3MB (3.3%)	230.6MB (26.9%)
GF	30.3MB (4.7%)	233.4MB (27.2%)
ALL	77.8MB (9.1%)	383.3MB (44.6%)

Table 4: Mercurial activity space overheads. All the overheads shown are computed as a percentage of the data in Ext2 (859MB).

porary file to the file specified in the patch. At one point during development, we moved from a Linux 2.6.19.1 kernel to a Linux 2.6.23.17 kernel. This resulted in a large patch touching all the source files in the repository. This forced a single instance of `patch` to read and write all of the 20,000 files in the Linux source tree. Every time `patch` writes to a new file, GF verifies that the file does not form a causality-violating cycle with the files that `patch` previously read. This results in the heavy system time overheads. This problem could be alleviated by having `patch` spawn multiple processes each of which merges a unique subset of the files specified in the patch file.

Another anomaly is that CA generates less causal data than OC. The explanation is that this workload generates a rename for every file to be patched. OC issues a freeze on the directory every time a directory is modified. CA, however, uses the causal history to determine that the same process is modifying the directory and eliminates duplicate entries. This results in CA consuming the least amount of both causal and version data.

Blast Workload Results Figure 8 shows the elapsed time results for the blast workload and Table 5 shows the space overhead. The overheads for this workload follow the pattern seen in the previous workloads. The time

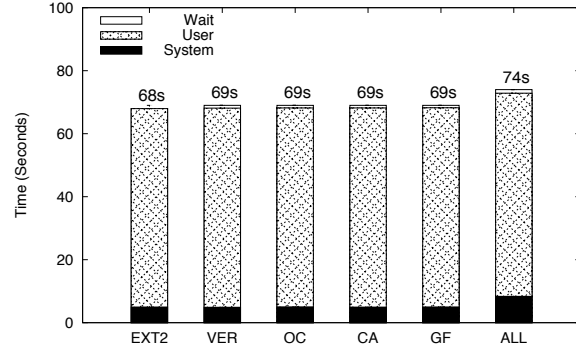


Figure 8: Blast elapsed time results.

	Causal Data	Version Space
VER	-	40KB (0.7%)
OC	172KB (2.9%)	40KB (0.7%)
CA	176KB (3.1%)	40KB (0.7%)
GF	172KB (2.9%)	36KB (0.6%)
ALL	3.7MB (65.4%)	14.4MB (257.4%)

Table 5: Blast workload space overheads. All the overheads shown are computed as a percentage of the data in Ext2 (5.8MB).

overhead is 1.4% for the VER, OC, CA, and GF configurations. The causal data overhead is less than 3.1% and the version data overhead is less than 1% for VER, OC, CA, and GF configurations. The workload is CPU intensive and processes a small number of large files, resulting in the minimal overheads for VER, OC, CA, and GF. For ALL, the elapsed time overhead is 8.8% and the space overhead is 65.4% on causal data and 257.4% on version data. The version data overheads of ALL is due to the behaviour of `formatdb` and `blast`. They write data in chunks smaller than a page, which versions the same page multiple times.

6.2 Recovery Benchmark Results

The goal of this subsection is to answer the following questions: First, in scenarios where open-close is sufficient (such as the first three use cases in Section 3), do the (unnecessary) causality-based algorithms impose additional recovery overhead? Second, in scenarios where causality does matter (the last two use cases in Section 3), how do the algorithms compare in recovery time and data loss?

To answer the first question, we wrote a program that simulates the behaviour of a worm. Worms typically overwrite one or more files/executables (for example, common executables like `ls`) and install some new

	Causal Data	Version data	Bytes Read			Recovery Time		
			11 th	7 th	3 rd	11 th	7 th	3 rd
OC	60KB	12KB	-	-	-	-	-	-
CA	176KB	470.5MB	39.15MB	39.16MB	39.17MB	23s	25.2s	27.4s
GF	184KB	470.5MB	39.15MB	39.16MB	39.15MB	23s	24.6s	25.8s
ALL	76.9MB	1.97GB	45.24MB	53.99MB	62.76	214.2s	452.8s	689s

Table 7: Results for recovering from the Apache simulator. All algorithms recover the same amount of data (40MB), but read in different amounts of data to perform the recovery.

	Causal Data	Version Space	Recovery Time	Bytes Read
OC	21.2MB	151.6MB	173.4s	179.1MB
CA	12.7MB	150.4MB	163.2s	163.9MB
GF	21.1MB	151.9MB	173.2s	179.1MB
ALL	52.8MB	249.3MB	191.2s	182.9MB

Table 6: Results for recovering from a worm attack. All algorithms recover the same amount of data (161.68MB), but read different amounts of data to perform the recovery.

files/programs (irchat servers being a popular choice). Our worm-simulator functions in a similar manner. The program traverses a copy of the Linux-2.6.19.1 source tree, overwrites some files and creates new “bad” files. All in all, we taint 25,600 files, writing a total of 500MB of data. Table 6 shows the time taken for recovering from this attack by each of the versioning algorithms. Recovery is performed in two phases. In the first phase, once a malicious process has been identified, we traverse up that process’s causal data graph to determine the root cause of the break-in. Backtracker [11] and Taser [7] perform a similar analysis to determine the cause. In the second phase, once we know the root cause of the attack, we propagate down the root process descendant tree to identify potential victims and recover them to a version just before the malicious process tampered with it. The recovery times that we report here are the times of the second phase. The results show that the recovery times are proportional to the amount of causal and versioning data stored. CA has the best recovery time and ALL has the worst recovery time. This is despite the fact ALL recovers the same amount of data as other algorithms and reads roughly the same amount of data as OC and GF to perform the recovery. ALL stores more versioning information than the other algorithms. Hence the required recovery data is spread over a much larger area on the disk. In turn, the recovery process has to perform more seeks to recover the same amount of data.

To answer the second question, we wrote a benchmark that simulates the Apache vulnerability scenario described in Section 1. The program first creates 50

files and then performs the following action in a loop 50 times. In each loop it writes 8KB to each file from start to end. Every n^{th} iteration (4^{th} in our implementation), the program forks a helper program that reads a byte from each of the 50 files and communicates the character to the main program via a pipe. This simulates the behaviour of a web server opening a new connection on a socket. In the causality based algorithms, once the main process reads from the pipe, it is a causally different version as it has read data from a new source, i.e., a new process. Hence any writes the main process performs after that creates a new version of the file. For this workload, OC does not copy any data in its version files; all the files were just created, so it considers all the writes to happen to version 1 of the files. CA and GF copy data to the version files on the iterations during which the the parent spawns a child and reads from the pipe. This stores 470.5MB of version data. The ALL algorithm copies data on every write and this adds up to around 2GB of data. The amount of version data is shown in Table 7.

We then recover versions at various intervals to get a sense for how expensive it is to go back further in time in each algorithm. There are 12 causality events in all, corresponding to the number of times a child is forked. We measure the time taken to recover data to a state before the 11th, 7th, and the 3rd event. The 11th corresponds to recovery close to the latest version, the 7th corresponds to recovery two thirds of the way back, and the 3rd corresponds to recovery close to one third of the way back. The results of this benchmark are shown in Table 7. With OC, there are no intermediate versions, so it cannot recover anything useful. CA and GF can both recover to a correct version and they both take the same amount of time to recover. They also read only the exact amount of data to be recovered. ALL, however, takes at least 9 times longer than CA and GF to perform recovery, because it has more data and has to search through a large amount of data to rebuild the correct version. Further, ALL has many more false positives that it has to filter before deciding on the version to recover. CA and GF have only one version to choose. Since ALL has been versioning continuously, one version of a process has multiple

children. For example, the 3rd causal event has 41,000 children from which it has to narrow down to 50 versions. CA and GF have only 50 children for that causal event. Note that the numbers presented in the table do not include the time used to identify the version to recover.

6.3 Results Summary

In most cases, CA introduces little overhead relative to OC, yet it provides versions in cases where OC fails to do so. GF performs comparably in many cases, but sometimes imposes high run time overheads. Version-on-every-write practically always performs poorly both in terms of space and elapsed time. For recovery, however, CA and GF can indeed be a big win in terms of both the time to recover a particular version and the amount of data lost.

7 Related work

Several prior research projects have built versioning systems. We categorize these systems by the versioning algorithm that they use and discuss each class in turn. We begin with the version-on-every-write systems. CVFS [24] was designed with security in mind. Each individual write or small metadata change (e.g., atime updates) is versioned. The research focuses on methods to store and access old versions efficiently. We adopted the CVFS approach of using a journal to store old version data. Wayback [3] is a user-level versioning file system built on the FUSE framework. On a `write` call, Wayback logs the data being overwritten to an undo log before completing the write. Our version file format is similar to that of Wayback, but Wayback versions on every write while we version more selectively. The Repairable file system (RFS) [29] has functionality closest to ours. They record both causal data and save versions. They, however, collect causal data and data blocks separately, thus preventing them from taking advantage of causal information to version more selectively, leading to versioning on every write. They also have to reconcile the causal and versioning data using timestamps as they collect them separately.

Now we discuss systems that use open-close versioning. Elephant [18] is a versioning file system implemented in the FreeBSD 2.2.8 kernel. Their research focus is on providing users with a range of version *retention* policies. Versionfs [10] is a stackable versioning file system. Versionfs allows users to selectively version files and is focused on the ability to set space reclamation and version storage policies for files. Retention/reclamation policies are complementary to our work.

As we discussed in section 1, snapshots are another approach for versioning where an image of a file system is

made periodically. Systems with snapshot functionality include AFS [13], Plan-9 [16], WAFL [9], [6], Venti [17], Ext3COW [15], Thresher [21], and Selective versioning secure disk system [27]. Skippy [20] proposes metadata indexing schemes that can be used to quickly lookup previous snapshots of a database.

Several systems have used causal data to provide various functions. The Taser intrusion recovery system [7] logs all system calls and their arguments. In the event of administrative errors or intrusions, they perform causal analysis on the logged data to determine the actions that need to be done to recover the system. They explore various algorithms and policies that can be used to determine the exact operations to be performed during recovery. We can leverage all of these algorithms and policies in our work, applying them in an online setting. As future work, they plan to integrate their work with versioning file systems to reduce the disk space requirements and to improve scalability. Our work has continued where Taser stopped and has taken a step further by integrating both causal and versioning systems. BackTracker [11] logs all system calls and in the event of an intrusion, performs causality analysis to determine the root cause of an intrusion. Autobash [26] is a configuration debugging tool that leverages causal information to limit the amount of testing required.

Chapman et.al. [5], explore techniques for causal data pruning. Their approach for pruning is to remove duplicates (which we already perform) and factor out common subtrees in causal graphs. Another approach for pruning causality could be to merge the causal information of deleted temporary files into their causal ancestors. Space can also be reclaimed by deleting the versioning data of temporary files, where temporary files are intermediate nodes in a causal graph.

Finally, a number of versioning algorithms have been explored by the object oriented database (OODB) community. These algorithms are focused on aspects that are particular to OODBs such as “how to propagate version changes of sub objects to composite objects?”, “how to present a consistent view in the face of updates to different objects?” [4], “how to version classes as they change” [28], etc.

8 Conclusions

Combining versioning and causal relationship data offers powerful capabilities above and beyond what each kind of system can do in isolation. Causality-based versioning ensures that we create meaningful versions of objects, facilitating better recovery from data-corrupting activities under concurrent workloads. While versioning introduces overheads between 1% and 25%, adding causal collection on top of versioning adds only an additional

5–6% overhead. The Cycle-Avoidance algorithm, which restricts itself to considering only per-object, local information during online operation provides superior versioning and recovery, at cost comparable to open-close.

Providing versioning in the context of PASS opens up future research possibilities in the areas of reproducibility and archival. PASS did not previously provide the ability to reproduce objects on the system, because they do not preserve all the necessary data. However, with versioning, the necessary data do exist. Versioning also produces objects that can easily be archived, and PASS provides the provenance to accurately describe those objects.

9 Acknowledgments

We thank Ethan Miller, our shepherd, and Margo Seltzer for repeated careful and thoughtful reviews of our paper. We thank Erez Zadok, Shankar Pasupathy, Jonathan Ledlie, and Uri Braun for their feedback on early drafts of the paper. We also thank Uri for validating the CA algorithm in our user level simulator. We thank the FAST reviewers for the valuable feedback they provided. This work was partially made possible thanks to NSF grant CNS-0614784.

References

- [1] ALTSCHUL, S. F., GISH, W., MILLER, W., MYERS, E. W., AND LIPMAN, D. J. Basic local alignment search tool. *Molecular Biology* 215 (1990), 403–410.
- [2] BRAUN, U., GARFINKEL, S., MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., AND SELTZER, M. Issues in automatic provenance collection. In *Proceedings of the 2006 International Provenance and Annotation Workshop* (May 2006).
- [3] BRIAN CORNELL AND PETER DINDA AND FABIN BUSTAMANTE. Wayback: A User-level Versioning File System for Linux. In *Proceedings of the USENIX 2004 Annual Technical Conference, FREENIX Track* (2004).
- [4] CELLARY, W., AND JOMIER, G. Consistency of versions in objects-oriented databases. In *Proceedings of the Sixteenth International Conference on Very Large Databases* (1990).
- [5] CHAPMAN, A. P., JAGADISH, H. V., AND RAMANAN, P. Efficient provenance storage. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), ACM, pp. 993–1006.
- [6] CHUTANI, S., ANDERSON, O. T., KAZAR, M. L., LEVERETT, B. W., MASON, W. A., AND SIDEBOTHAM, R. N. The Episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference* (San Francisco, CA, 1992), pp. 43–60.
- [7] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The Taser intrusion recovery system. In *SOSP* (2005).
- [8] HALCROW, M. A. eCryptfs: An enterprise-class encrypted filesystem for linux. *Ottawa Linux Symposium* (2005).
- [9] HITZ, D., LAU, J., AND MALCOLM, M. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference* (January 1994), pp. 235–245.
- [10] K. MUNISWAMY-REDDY AND C. P. WRIGHT AND A. HIMMER AND E. ZADOK. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)* (March/April 2004).
- [11] KING, S. T., AND CHEN, P. M. Backtracking Intrusions. In *SOSP* (Bolton Landing, NY, October 2003).
- [12] KING, S. T., MAO, Z. M., LUCCHETTI, D. G., AND CHEN, P. M. Enriching intrusion alerts through multi-host causality. In *the 12th Annual Network and Distributed System Security Symposium* (2005).
- [13] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles* (1991).
- [14] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*.
- [15] PETERSON, Z., AND BURNS, R. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage* 1, 2 (2005), 190–212.
- [16] QUINLAN, S. A Cached WORM File System. *Software – Practice and Experience* 21, 12 (1991), 1289–1299.
- [17] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *Proceedings of First USENIX conference on File and Storage Technologies* (January 2002), pp. 89–101.
- [18] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R., AND OFIR, J. Deciding When to Forget in the Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (December 1999).
- [19] SHAH, S., SOULES, C. A. N., GANGER, G. R., AND NOBLE, B. D. Using provenance to aid in personal file search. In *Proceedings of the USENIX Annual Technical Conference* (2007).
- [20] SHAULL, R., SHRIRA, L., AND XU, H. Skippy: a new snapshot indexing method for time travel in the storage manager. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA).
- [21] SHRIRA, L., AND XU, H. Thresher: An efficient storage manager for copy-on-write snapshots. In *Proceedings of the Usenix Annual Technical Conference* (Boston, MA, May 2006).
- [22] SIMMHAN, Y. L., PLALE, B., AND GANNON, D. A survey of data provenance in e-science. *SIGMOD Rec.* 34, 3 (2005), 31–36.
- [23] SOMAYAJI, A., AND FORREST, S. Automated Response Using System-Call Delays. In *USENIX Security Symposium* (2000).
- [24] SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. Metadata Efficiency in Versioning File Systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (March 2003), pp. 43–58.
- [25] Apache httpd 1.3 vulnerabilities. http://httpd.apache.org/security/vulnerabilities_13.html.
- [26] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. Autobash: improving configuration management with operating system causality analysis. In *SOSP '07: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), ACM, pp. 237–250.
- [27] SUNDARARAMAN, S., SIVATHANU, G., AND ZADOK, E. Selective versioning in a secure disk system. In *Proceedings of the 17th USENIX Security Symposium* (July-August 2008).
- [28] TALENS, G., OUSSALAH, C., AND COLINAS, M. F. Versions of simple and composite objects. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1993).
- [29] ZHU, N., AND CHIUEH, T.-C. Design, implementation, and evaluation of repairable file service. In *The International Conference on Dependable Systems and Networks* (2003).

Enabling Transactional File Access via Lightweight Kernel Extensions

Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, and Erez Zadok
Stony Brook University

Charles P. Wright
IBM T.J. Watson Research Center

Abstract

Transactions offer a powerful data-access method used in many databases today through a specialized query API. User applications, however, use a different file-access API (POSIX) which does not offer transactional guarantees. Applications using transactions can become simpler, smaller, easier to develop and maintain, more reliable, and more secure. We explored several techniques how to provide transactional file access with minimal impact on existing programs. Our first prototype was a standalone kernel component within the Linux kernel, but it complicated the kernel considerably and duplicated some of Linux's existing facilities. Our second prototype was all in user level, and while it was easier to develop, it suffered from high overheads. In this paper we describe our latest prototype and the evolution that led to it. We implemented a transactional file API inside the Linux kernel which integrates easily and seamlessly with existing kernel facilities. This design is easier to maintain, simpler to integrate into existing OSs, and efficient. We evaluated our prototype and other systems under a variety of workloads. We demonstrate that our prototype's performance is better than comparable systems and comes close to the theoretical lower bound for a log-based transaction manager.

1 Introduction

In the past, providing a transactional interface to files typically required developers to choose from two undesirable options: (1) modify complex file system code in the kernel or (2) provide a user-level solution which incurs unnecessary overheads. Previous in-kernel designs either had the luxury of designing around transactions from the beginning [33] or limited themselves to supporting only one primary file system [43]. Previous user-level approaches were implemented as libraries (e.g., Berkeley DB [39], and Stasis [34]) and did not support interaction through the VFS [15] with other non-transactional processes. These libraries also introduced a redundant page cache and provided no support to non-transactional processes. This paper presents the design and evaluation of a transactional file interface that requires modifications to neither existing file systems nor applications, yet guarantees atomicity and isolation for

standard file accesses using the kernel's own page cache.

Transactions require satisfaction of the four ACID properties: Atomicity, Consistency, Isolation, and Durability. Enforcing these properties appears to require many OS changes, including a unified cache manager [12] and support for logging and recovery. Despite the complexity of supporting ACID semantics on file operations [30], Microsoft [43] and others [4, 44] have shown significant interest in transactional file systems. Their interest is not surprising: developers are constantly reimplementing file cleanup and ad-hoc locking mechanisms which are unnecessary in a transactional file system. A transactional file system does not eliminate the need for locking and recovery, but by exposing an interface to specify transactional properties allows application programmers to reuse locking, logging, and recovery code. Defending against TOCTTOU (time of check till time of use) security attacks also becomes easier [28, 29] because sensitive operations are easily isolated from an intruder's operations. Security and quality guarantees for control files, such as configuration files, are becoming more important. The number of programs running on a standard system continues to grow along with the cost of administration. In Linux, the CUPS printing service, the Gnome desktop environment, and other services all store their configurations in files that can become corrupted when multiple writers access them or if the system crashes unexpectedly. Despite the existence of database interfaces, many programs still use configuration files for their simplicity, generality, and because a large collection of existing tools can access these simple configuration files. For example, Gnome stores over 400 control files in a user's home directory. A transactional file interface is useful to all such applications.

To provide ACID guarantees, a file interface must be able to mediate all access to the transactional file system. This forces the designer of a transactional file system to put a large database-like runtime environment either in the kernel or in a kernel-like interceptor, since the kernel typically services file-system system calls. This environment must employ abortable logging and recovery mechanisms that are linked into the kernel code. VFS-cache rollback is also required to revert an aborted transaction [44], its stale inodes, dentries, and other in-kernel data structures. The situation can be simplified drasti-

cally if one abandons the requirement that the backing store for file operations must be able to interact with other transaction-oblivious processes (e.g., `grep`), and by duplicating the functionality of the page cache in user space. This concession is often made by transactional libraries such as Berkeley DB [39] and Stasis [34]: they provide a transactional interface only to a single file and they do not solve the complex problems of rewinding the page cache and stale in-memory structures after a process aborts. Systems such as QuickSilver [33] and TxF [43] address this trade-off between the completeness and implementation size by redesigning a specific file system around proper support for transactional file operations. In this paper we show that such a redesign is unnecessary, and that every file system can provide a transactional interface without requiring specialized modifications. We describe our system which uses a seamless approach to provide transactional semantics using a new dynamically loaded kernel module, and only minor modifications to existing kernel code. Our technique keeps kernel complexity low yet still offers a full-fledged transactional file interface without introducing unnecessary overheads for non-transactional processes.

We call our file interface *Valor*. Valor relies on improved locking and write ordering semantics that we added to the kernel. Through a kernel module, it also provides a simple in-kernel logging subsystem optimized for writing data. Valor's kernel modifications are small and easily separable from other kernel components; thus introducing negligible kernel complexity. Processes can use Valor's logging and locking interfaces to provide ACID transactions using seven new system calls. Because Valor enforces locking in the kernel, it can protect operations that a transactional process performs from any other process in the system. Valor aborts a process's transaction if the process crashes. Valor supports large and long-living transactions. This is not possible for `ext3`, `XFS`, or any other journaling file system: these systems can only abort the entire file system journal, and only if there is a hardware I/O error or the entire system crashes. These systems' transactions must always remain in RAM until they commit (see Section 2).

Another advantage of our design is that it is implemented on top of an unmodified file system. This results in negligible overheads for processes not using transactions: they simply access the underlying file system, only using the Valor kernel modifications to acquire necessary locks. Using tried-and-true file systems also provides good performance compared to systems that completely replace the file system with a database. Valor runs with a statistically indistinguishable overhead on top of `ext3` under typical loads when providing a transactional interface to a number of sensitive configuration files. Valor is designed from the beginning to run well

without durability. File system semantics accept this as the default, offering `fsync(2)` [9] as the accepted means to block until data is safely written to disk. Valor has an analogous function to provide durable commits. This makes sense in a file-system setting as most operations are easily repeatable. For non-durable transactions, Valor's overhead on top of an idealized mock logging implementation is only 35% (see Section 4).

The rest of this paper is organized as follows. In Section 2 we describe previous experiences with designing transactional systems and related work that have led us to Valor. We detail Valor's design in Section 3 and evaluate its performance in Section 4. We conclude and propose future work in Section 5.

2 Background

The most common approach for transactions on stable storage is using a relational database, such as an SQL server (e.g., MySQL [22]) or an embedded database library (e.g., Berkeley DB [39]); but they have also long been a desired programming paradigm for file systems. By providing a layer of abstraction for concurrency, error handling, and recovery, transactions enable simpler, more robust programs. Valor's design was informed by two previous file systems we developed using Berkeley DB: `KDBFS` and `Amino` [44]. Next we discuss journaling file systems' relationship to our work, and we follow with discussions on database file systems and APIs.

2.1 Beyond Journaling File Systems

Journaling file systems suffer from two draw-backs: (1) they must store all data modified by a transaction in RAM until the transaction commits and (2) their journals are not designed to be accessed by user processes [16, 31, 42]. Journaling file systems store only enough information to commit a transaction already stored in the log (redo-only record). This results in journaling file systems being forced to contain all data for all in-flight transactions in RAM [6, 7, 42]. For metadata transactions, which are finite in size and duration, journaling file systems are a convenient optimization. However, we wanted to provide user processes with transactions that could be megabytes large and run for long periods of time. The RAM restriction of a journaling file system is too limiting to support versatile file-based transactions.

Two primary approaches were used to provide file-system transactions to user processes. (1) *Database file systems* provide transactions to user processes by making fundamental changes to the design of a standard file system to support better logging and rollback of inodes, dentries, and cached pages [33, 36, 43]. (2) *Database access APIs* provide transactions to user processes by offering a user library that exposes a transactional page file. Processes can store application data in the page file

by using library-specific API routines rather than storing their data on the file system [34, 39]. Valor represents an alternative to the above two approaches. Valor's design was settled after designing KBDBFS and Amino [44]. We discuss KBDBFS and Amino in their proper contexts in Sections 2.2 and 2.3, respectively.

2.2 Database File Systems

KBDBFS was an in-kernel file system built on a port of the Berkeley Database [39] to the Linux kernel. It was part of a larger project that explored uses of a relational database within the kernel. KBDBFS utilized transactions to provide file-system-level consistency, but did not export these same semantics to user-level programs. It became clear to us that unlocking the potential value of a file system built on a database required exporting these transactional semantics to user-level applications. KBDBFS could not easily export these semantics to user-level applications, because as a standard kernel file system in Linux it was bound by the VFS to cache various objects (e.g., inodes and directory entries), all of which ran the risk of being rolled back by the transaction. To export transactions to user space, KBDBFS would therefore be required to either bypass the VFS layers that require these cached objects, or alternatively track each transaction's modifications to these objects. The first approach would require major kernel modifications and the second approach would duplicate much of the logging that BDB was already providing, losing many of the benefits provided by the database.

Our design of KBDBFS was motivated in part by a desire to modify the existing Linux kernel as little as possible. Another transaction system which modified an existing OS was Seltzer's log-structured file system, modified to support transaction processing [37]. Seltzer et al's simulations of transactions embedded in the file system showed that file system transactions can perform as well as a DBMS in disk-bound configurations [35]. They later implemented a transaction processing (TP) system in a log-structured file system (LFS), and compared it to a user-space TP system running over LFS and a read-optimized file system [37].

Microsoft's TxF [19,43] and QuickSilver's [33] database file systems leverage the early incorporation of transactions support into the OS. TxF exploits the transaction manager which was already present in Windows. TxF uses multiple file versions to isolate transactional readers from transactional writers. TxF works only with NTFS and relies on specific NTFS modifications and how NTFS interacts with the Windows kernel. QuickSilver is a distributed OS developed by IBM Research that makes use of transactional IPC [33]. QuickSilver was designed from the ground up using a microkernel architecture and IPC. To fully integrate transactions into

the OS, QuickSilver requires a departure from traditional APIs and requires each OS component to provide specific rollback and commit support. We wanted to allow existing applications and OS components to remain largely unmodified, and yet allow them to be augmented with simple begin, commit, and abort calls for file system operations. We wanted to provide transactions without requiring fundamental changes to the OS, and without restricting support to a particular file system, so that applications can use the file system most suited to their work load on any standard OS. Lastly, we did not want to incur any overheads on non-transactional processes.

Inversion File System [24], OdeFS [5], iFS [26], and DBFS [21] are database file systems implemented as user-level NFS servers [17]. As they are NFS servers (which predate NFSv4's locking and callback capabilities [38]), the NFS client's cache can serve requests without consulting the NFS server's database; this could allow a client application to write to a portion of the file system that has since been locked by another application, violating the client application's isolation. They do not address the problem of supporting efficient transactions on the local disk.

2.3 Database Access APIs

The other common approach to providing a transactional interface to applications is to provide a user-level library to store data in a special page file or B-Tree maintained by the library. Berkeley DB offers a B-Tree, a hash table, and other structures [39]. Stasis offers a page file [34]. These systems require applications to use database-specific APIs to access or store data in these library-controlled page files.

Based on our experiences with KBDBFS, we chose to prototype a transactional file system, again built on BDB, but in user space. Our prototype, Amino, utilized Linux's process debugging interface, `ptrace` [8], to service file-system-related calls on behalf of other processes, storing all data in an efficient Berkeley DB B-tree schema. Through Amino we demonstrated two main ideas. First, we revealed the ability to provide transactional semantics to user-level applications. Second, we showed the benefits that user-level programs gain when they use these transactional semantics: programming model simplification and application-level consistency [44]. Although we extended `ptrace` to reduce context switches and data copies, Amino's performance was still poor compared to an in-kernel file system for some system-call-intensive workloads (such as the configuration phase of a compile). Finally, although Amino's performance was comparable to Ext3 for metadata workloads (such as Postmark [14]), for data-intensive workloads, Amino's database layout resulted in significantly lower throughput. Amino was a

successful project in that it validated the concept of a transactional file system with a user-visible transactional API, but the performance we achieved could not displace traditional file systems. Moreover, one of our primary goals is for transactional and non-transactional programs to have access to the same data through the file system interface. Although Amino provided binary compatibility with existing applications, running programs through a `ptrace` monitor is not as seamless as we liked. The `ptrace` monitor had to run in privileged mode to service all processes, it serviced system calls inefficiently due to additional memory copies and context switches, and it imposed additional overhead from using signal passing to simulate a kernel system call interface for applications [44]. Other user level approaches to providing transactional interfaces include Berkeley DB and Stasis.

Berkeley DB. Berkeley DB is a user library that provides applications with an API to transactionally update key-value pairs in an on-disk B-Tree. We discuss Berkeley DB's relative performance in depth in Section 4. We benchmark BDB through Valor's file system extensions. Relying on BDB to perform file system operations can result in large overheads for large serial writes or large transactions (256MiB or more). This is because BDB is being used to provide a file interface, which is used by applications with different work-loads than applications that typically use a database. If the regular BDB interface is used, though, transaction-oblivious processes cannot interact with transactional applications, as the former use the file system interface directly.

Stasis. Stasis provides applications a transactional interface to a page file. Stasis requires that applications specify their own hooks to be used by the database to determine efficient undo and redo operations. Stasis supports nested transactions [7] alongside write-ahead logging and LSN-Free pages [34] to improve performance. Stasis does not require applications to use a B-Tree on disk and exposes the page file directly. Like BDB, Stasis requires applications to be coded against its API to read and write transactionally. Like BDB, Stasis does not provide a transactional interface on top of an existing file system which already contains data. Also like BDB, Stasis implements its own private, yet redundant page cache which is less efficient than cooperating with the kernel's page cache (see Section 4).

Reflecting on our experience with KBDBFS and Amino, we have come to the conclusion that adapting the file system interface to support ACID transactions does indeed have value and that the two most valuable properties that the database provided to us were the logging and the locking infrastructure. Therefore, in Valor we provide two key kernel facilities: (1) extended mandatory locking and (2) simple write order-

ing. Extended mandatory locking lets Valor provide the isolation that in our previous prototypes was provided by the database's locking facility. Simple write ordering lets Valor's logging facility use the kernel's page cache to buffer dirty pages and log pages which reduces redundancy, improves performance, and makes it easier to support transactions on top of existing file systems.

3 Design and Implementation

The design of Valor prioritizes (1) a low complexity kernel design, (2) a versatile interface that makes use of transactions optional, and (3) performance. Our seamless approach achieves low complexity by exporting just a minimal set of system calls to user processes. Functionality exposed by these system calls would be difficult to implement efficiently in user-space.

Valor allows applications to perform file-system operations within isolated and atomic transactions. Isolation guarantees that file-system operations performed within one transaction have no impact on other processes. Atomicity guarantees that committing a transaction causes all operations performed in it to be performed at once, as a unit inseparable even by a system crash. If desired, Valor can ensure a transaction is durable: if the transaction completes, the results are guaranteed to be safe on disk. We now turn to Valor's *transactional model*, which specifies the scope of these guarantees and what processes must do to ensure they are provided.

Transactional Model. Valor's transactional guarantees extend to the individual inodes and pages of directories and regular files for reads and writes. A process must lock an entire file if it will read from or write to its inode. Appends and truncations modify the file size, so they also must lock the entire file. To overwrite data in a file, only the affected pages need to be locked. When performing directory operations like file creation and unlinking, only the containing directory needs to be locked. When renaming a directory, processes must also recursively lock all of the directory's descendants. This is the accepted way to handle concurrent lockers during a directory rename [27]. More sophisticated locking schemes (e.g., intent locks [3]) that improve performance and relieve contention among concurrent processes are beyond the scope of this paper.

We now turn to the concepts underlying Valor's architecture. These concepts are implemented as components of Valor's system; they are illustrated in Figure 1.

1. Logging Device. In order to guarantee that a sequence of modifications to the file system completes as a unit, Valor must be able to undo partial changes left behind by a transaction that was interrupted by either a system crash or a process crash. This means that Valor

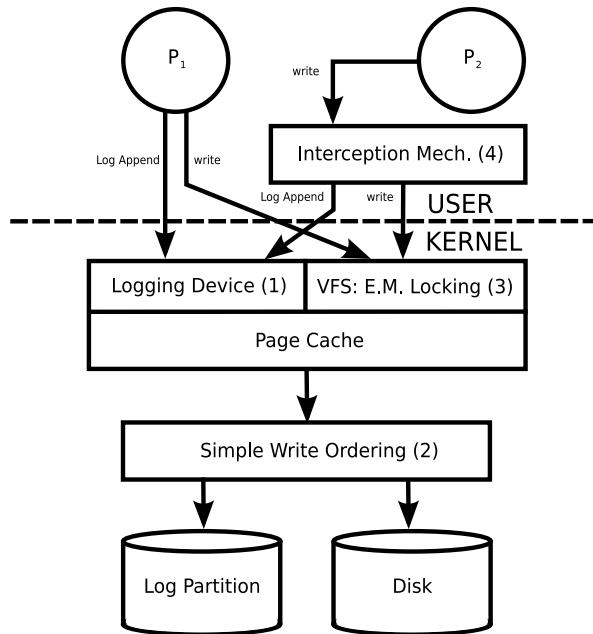


Figure 1: Valor Architecture

must store some amount of auxiliary data, because an unmodified file system can only be relied upon to atomically update a single sector and does not provide a mechanism for determining the state before an incomplete write. Common mechanisms for storing this auxiliary data include a *log* [7] and WAFL [13]. Valor does not modify the existing file system, so it uses a log stored on a separate partition called the *log partition*.

2. Simple Write Ordering. Valor relies on the fact that even if a write to the file system fails to complete, the auxiliary information has already been written to the log. Valor can use that information to undo the partial write. In short, Valor needs to have a way to ensure that writes to the log partition occur before writes to other file systems. This requirement is a special case of *write ordering*, in which the page cache can control the order in which its writes reach the disk. We discuss our implementation in Section 3.1, which we call *simple write ordering* both because it is a special case and because it operates specifically at page granularity.

3. Extended Mandatory Locking. Isolation gives a process the illusion that there are no other concurrently executing processes accessing the same files, directories, or inodes. Transactional processes can implement this by first acquiring a lock before reading or writing to a page in a file, a file's inode, or a directory. However, an OS with a POSIX interface and pre-existing applications must support processes that do not use transactions. These *transaction-oblivious* processes do not acquire locks before reading from or writing to files or directories. *Extended mandatory locking* ensures that

all processes acquire locks before accessing these resources. See Section 3.2.

4. Interception Mechanism. New applications can use special APIs to access the transaction functionality that Valor provides; however, pre-existing applications must be made to run correctly if they are executed inside a transaction. This could occur if, for example, a Valor-aware application starts a transaction and launches a standard shell utility. To do this, Valor modifies the standard POSIX system calls used by unmodified applications to perform the locking necessary for proper isolation. Section 3.3 describes our modifications.

The above four Valor components provide the necessary infrastructure for the seven Valor system calls. Processes that desire transactional semantics must use the Valor system calls to log their writes and acquire locks on files. We now discuss the Valor system calls and then provide a short example to illustrate Valor's basic operation.

Valor's Seven System Calls. When an application uses the following seven system calls correctly (e.g., calling the appropriate system call before writing to a page), Valor provides that application fully transactional semantics. This is true even if other user-level applications do not use these system calls or use them incorrectly.

Log Begin begins a transaction. This must be called before all other operations within the transaction.

Log Append logs an *undo-redo record*, which stores the information allowing a subsequent operation to be reversed. This must be called before every operation within the transaction. See Section 3.1.

Log Resolve ends a transaction. In case of an error, a process may voluntarily *abort* a transaction, which undoes partial changes made during that transaction. This operation is called an *abort*. Conversely, if a process wants to end the transaction and ensure that changes made during a transaction are all done as an atomic unit, it can *commit* the transaction. Whether a *log resolve* is a commit or an abort depends on a flag that is passed in.

Transaction Sync flushes a transaction to disk. A process may call *Transaction Sync* to ensure that changes made in its committed transactions are on disk and will never be undone. This is the only sanctioned way to achieve durability in Valor. `O_DIRECT`, `O_SYNC`, and `fsync` [9] have no useful effect within a transaction for the same reason that nested transactions cannot be durable: the parent transaction has yet to commit [7].

Lock, Lock Permit, Lock Policy Our *Lock* system call locks a page range in a file, an entire directory, or an entire file with a shared or exclusive

lock. This is implemented as a modified `fcntl`. These routines provide Valor's support for transactional isolation. `Lock Permit` and `Lock Policy` are required for security and inter-process transactions, respectively. See Section 3.2.

Cooperating with the Kernel Page Cache. As illustrated in Figure 1, the kernel's page cache is central to Valor, and one of Valor's key contributions is its close cooperation with the page cache. In systems that do not support transactions, the `write(2)` system call initiates an asynchronous write which is later flushed to disk by the kernel page cache's dirty-page write-back thread. In Linux, this thread is called `pdflush` [1]. If an application requires durability in this scenario, it must explicitly call `fsync(2)`. Omitting durability by default is an important optimization which allows `pdflush` to economize on disk seeks by grouping writes together. Databases, despite introducing transaction semantics, achieve similar economies through *No-Force* page caches. These caches write auxiliary log records only when a transaction commits, and then only as one large serial write, and use threads similar to `pdflush` to flush data pages asynchronously [7]. Valor is also *No-Force*, but can further reduce the cost of committing a transaction by writing nothing—neither log pages nor data pages—until `pdflush` activates. Valor's simple write ordering scheme facilitates this optimization by guaranteeing that writes to the log partition always occur before the corresponding data writes. In the absence of simple write ordering, Valor would be forced to implement a redundant page cache, as many other systems do. Valor implements simple write ordering in terms of existing Linux `fsync` semantics which returns when the writes are scheduled, but before they hit the disk platter. This introduces a short race where applications running on top of Valor and the other systems we evaluated (Berkeley DB, Stasis, and ext3) could crash unrecoverably. Unfortunately, this is the standard `fsync` implementation and impacts other systems such as MySQL, Berkeley DB, and Stasis [45] which rely on `fsync` or its like (i.e., `fdatasync`, `O_SYNC`, and `direct-IO`).

One complexity introduced by this scheme is that a transaction may be completely written to the log, and reported as durable and complete, but its data pages may not yet all be written to disk. If the system crashes in this scenario, Valor must be able to complete the disk writes during recovery to fulfill its durability guarantee. Similar to database systems that also perform this optimization, Valor includes sufficient information in the log entries to *redo* the writes, allowing the transaction to be completed during recovery.

Another complexity is that Valor supports large transactions that may not fit entirely in memory. This means

that some memory pages that were dirtied during an incomplete transaction may be flushed to disk to relieve memory pressure. If the system crashes in this scenario, Valor must be able to rollback these flushes during recovery to fulfill its atomicity guarantee. Valor writes *undo* records describing the original state of each affected page to the log when flushing in this way. A page cache that supports flushing dirty pages from uncommitted transactions is known as a *Steal* cache; XFS [41], ZFS [40], and other journaling file systems are *No-Steal*, which limits their transaction size [42] (see Section 2). Valor's solution is a variant of the ARIES transaction recovery algorithm [20].

An Example. Figure 2 illustrates Valor's writeback mechanism. A process P_2 initially calls the `Lock` system call to acquire access to two data pages in a file, then calls the `Log Append` system call on them, generating the two 'L's in the figure, and then calls `write(2)` to update the data contained in the pages, generating the two 'P's in the figure. Finally, it commits the transaction and quits. The processes did not call `transaction sync`. On the left hand side, the figure shows the state of the system before P_2 commits the transaction; because of Valor's non-durable *No-Force* logging scheme, data pages and corresponding undo/redo log entries both reside in the page cache. On the right hand side, the process has committed and exited; simple write ordering ensures that the log entries are safely resident on disk, and the data pages will be written out by `pdflush` as needed.

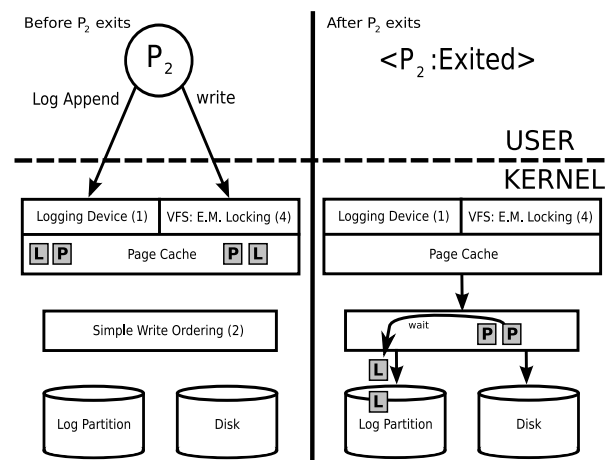


Figure 2: Valor Example

We now discuss each of Valor's four architectural components in detail. Section 3.1 discusses the logging, simple write ordering, and recovery components of Valor. Section 3.2 discusses Valor's extended mandatory locking mechanism, and Section 3.3 explains Valor's interception mechanism.

3.1 The Logging Interface

Valor maintains two logs. A *general-purpose log* records information on directory operations, like adding and removing entries from a directory, and inode operations, like appends or truncations. A *page-value log* records modifications to individual pages in regular files [2]. Before writing to a page in a regular file (*dirtying* the page), and before adding or removing a name from a directory, the process must call `Log Append` to prepare the associated undo-redo record. We refer to this undo-redo record as a *log record*. Since the bulk of file system I/O is from dirtying pages and not directory operations, we have only implemented Valor's page log for evaluation. Valor manages its logs by keeping track of the state of each transaction, and tracking which log records belong to which transactions.

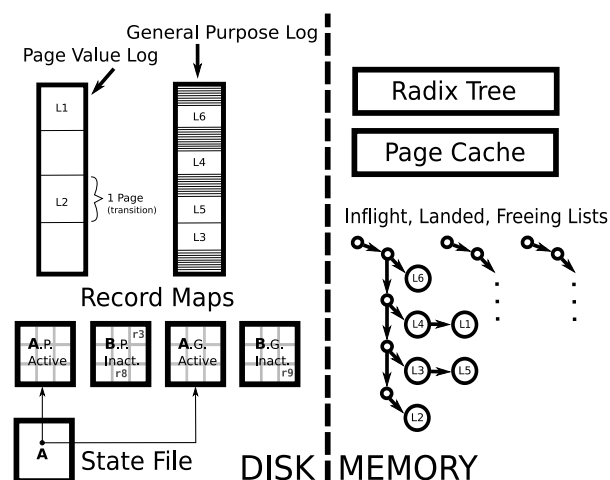


Figure 3: Valor Log Layout

3.1.1 In-Memory Data Structures

There are three states a transaction can be in during the course of its life: (1) *in-flight*, in which the application has called `Log Begin` but has not yet called `Log Resolve`; (2) *landed*, in which the application has called `Log Resolve` but the transaction is not yet safe to deallocate; and (3) *freeing*, in which the transaction is ready to be deallocated. Landed is distinct from freeing because if an application does not require durability, `Log Resolve` causes neither the log nor the data from the transaction to be flushed to disk (see above, *Cooperating with the Kernel Page Cache*).

Valor tracks a transaction by allocating a *commit set* for that transaction. A commit set consists of a unique *transaction ID* and a list of log records. As depicted in Figure 3, Valor maintains separate lists of in-flight, landed, and freeing commit sets. It also uses a radix tree to track free on-disk log records.

Life Cycle of a Transaction. When a process calls `Log Begin`, it gets a transaction ID by allocating a new log record, called a *commit record*. Valor then creates an in-memory commit set and moves it onto the inflight list. During the lifetime of the transaction, whenever the process calls `Log Append`, Valor adds new log records to the commit set. When the process calls `Log Resolve`, Valor moves its commit set to the landed list and marks it as *committed* or *aborted* depending on the flag passed in by the process. If the transaction is committed, Valor writes a *magic value* to the commit record allocated during `Log Begin`. If the system crashes and the log is complete, the value of this log record dictates whether the transaction should be recovered or aborted.

One thing Valor must be careful about is the case in which a log record is flushed to disk by `pdflush`, its corresponding file page is updated with a new value, and the file system containing that file page writes it to disk, thus violating write ordering. To resolve this issue, Valor keeps a flag in each page in the kernel's page cache. This flag can read *available* or *unavailable*; between the time Valor flushes the page's log record to the log and the time the file system writes the dirty page back to disk, it is marked as unavailable, and processes which try to call `Log Append` to add new log records wait until it becomes available, thus preserving our simple write ordering constraint. For hot file-system pages (e.g., those containing global counters), this could result in bursty write behavior. One possible remedy is to borrow Ext3's solution: when writing to an *unavailable* page, Valor can create a copy. The original copy remains read-only and is freed after the flush completes. The new copy is used for new reads and writes and is not flushed until the next `pdflush`, maintaining the simple write ordering.

We modified `pdflush` to maintain Valor's in-memory data structures and to obey simple write ordering by flushing the log's super block before all other super blocks. When `pdflush` runs, it (1) moves commit sets which have been written back to disk to the freeing list, (2) marks all page log records in the inflight and landed lists as unavailable, (3) atomically transitions the disk state to commit landed transactions to disk, and (4) iterates through the freeing list to deallocate transactions which have been safely written back to disk.

Soft vs. Hard Deallocations. Valor deallocates log records in two situations: (1) when a `Log Append` fails to allocate a new log record, and (2) when `pdflush` runs. *Soft deallocation* waits for `pdflush` to naturally write back pages and moves a commit set to the freeing list to be deallocated once all of its log records have had their changes written back. *Hard deallocation* explicitly flushes a landed commit set's dirty pages and directory modifications so it can immediately deallocate it.

3.1.2 On-Disk Data Structures

Figure 3 shows the page-value log and general-purpose log. Valor maintains two *record map* files to act as superblocks for the log files, and to store which log records belong to which transactions. One of these record map files corresponds to the general-purpose log, and the other to the page-value log. For a given log, there are exactly the same number of entries in the record map as there are log records in the log. The five fields of a record map entry are:

Transaction ID The transaction (commit set) this log record belongs to.

Log Sequence Number (LSN) Indicates when this log record was allocated.

inode Inode of the file whose page was modified.

netid Serial number of the device the inode resides on.

offset Offset of the page that was modified.

General-purpose log records contain directory path names for recovery of original directory listings in case of a crash. Page value log records contain a specially-encoded page to store both the undo and the redo record. The state file is part of the mechanism employed by Valor to ensure atomicity. It is described in Section 3.1.3 along with Valor's atomic flushing procedure.

Transition Value Logging. Although the undo-redo record of an update to a page could be stored as the value of the page before the update and the value after, Valor instead makes a reasonable optimization in which it stores only the XOR of the value of the page before and after the update. This is called a *transition page*. Transition pages can be applied to either recover or abort the on-disk image. A pitfall of this technique is that idempotency is lost [7]; Valor avoids this problem by recording the location and value of the first bit of each sector in the log record that differed between the undo and redo image. Although log records are always page-sized, this information must be stored on a per-sector basis as the disk may only write part of the page. (Because meta-data is stored in a separate map, transition pages in the log are all sector-aligned.) If a transaction updates the same page multiple times, Valor forces each `Log Append` call to wait on the Page Available flag which is set by the simple write ordering component operating within `pdflush`. If it does not have to wait, the call may update the log record's page directly, incurring no I/O. However, if the call must wait, then a new log record must be made to ensure recoverability.

3.1.3 LDST: Log Device State Transition

Valor's in-memory data structures are a reflection of Valor's on-disk state; however, as commit sets and log records are added, Valor's on-disk state becomes stale until the next time `pdflush` runs. We ensure that

`pdflush` performs an atomic transition of Valor's on-disk state to reflect the current in-memory state, thus making it no longer stale. To represent the previous and next state of Valor's on-disk files, we have a *stable* and *unstable* record map for each log file. The stable record maps serve as an authoritative source for recovery in the event of a crash. The unstable record maps are updated during Valor's normal operation, but are subject to corruption in the event of crashes. The purpose of Valor's LDST is to make the unstable record map consistent, and then safely and atomically relabel the stable record maps as unstable and vice versa. This is similar to the scheme employed by LFS [32, 37].

The core atomic operation of the LDST is a pointer update, in which Valor updates the state file. This file is a pointer to the pair of record maps that is currently stable. Because it is sector-aligned and less than one sector in size, a write to it is atomic. All other steps ensure that the record maps are accurate at the point in time where the pointer is updated. The steps are as follows:

1. Quiesce (block) all readers and writers to any on-disk file in the Valor partition.
2. Flush the inodes of the page-value and general-purpose log files. This flushes all new log records to disk. Log records can only have been added, so a crash at this point has no effect as the stable records map does not point to any of the new entries.
3. Flush the inodes of the unstable page-value and general-purpose record map files.
4. Write the names of the newly stable record maps to the state file.
5. Flush the inode of the state file. The up-to-date record map is now stable, and Valor now recovers from it in case of a system crash.
6. Copy the contents of the stable (previously unstable) record map over the contents of the unstable (previously stable) record map, bringing it up to date.
7. Un-quiesce (unblock) readers and writers.
8. Free all freeing log records.

Atomicity. The atomicity of transactions in Valor follows from two important constraints which Valor ensures that the OS obeys: (1) that writes to the log partition and data partitions obey simple write ordering and (2) that the LDST is atomic. At mount time, Valor runs recovery (Section 3.1.4) to ensure that the log is ready and fully describes the on-disk system state when it is finished mounting. Thereafter, all proper transactional writes are preceded by `Log Append` calls. No writes go to disk until `pdflush` is called or Valor's `Transaction Sync` is called. Simple write ordering ensures that in both cases, the log records are written before the in-place updates, so no update can reach the disk unless its

corresponding log record has already been written. Log records themselves are written atomically and safely because writes to the log's backing store are only made during an LDST. Since an LDST is atomic, the state of the entire system advances forward atomically as well.

3.1.4 Performing Recovery

System Crash Recovery. During the mount operation, the logging device checks to see if there are any outstanding log records and, if so, runs recovery. During umount, the Logging Device flushes all committed transactions to disk and aborts all remaining transactions. Valor can perform recovery easily by reading the state file to determine which record map for each log is stable, and reconstructing the commit sets from these record maps. A log sequence number (LSN) stored in the record map allows Valor to read in reverse order the events captured within the log and play them forward or back based on whether the write needs to be completed to satisfy durability or rolled back to satisfy atomicity. Recovery finds all record map entries and makes a commit set for each of them which is by default marked as aborted. While traversing through record map entries if it finds a record map entry with a magic value (written asynchronously during `Log Resolve`) indicating that this transaction was committed, it marks that set committed. Finally all commit sets are deallocated and an LDST is performed. The system can come on line.

Process Crash Recovery. Recovery handles the case of a system crash, something handled by all journaling file systems. However, Valor also supports user-process transactions and, by extension, user-process recovery. When a process calls the `do_exit` process clean-up routine in the kernel, their `task_struct` is checked to see if a transaction was in-flight. If so, then Valor moves the commit set for the transaction onto the landed list and marks the commit set as aborted.

3.2 Ensuring Isolation

Extended mandatory locking is a derivation of mandatory locking, a system already present in Linux and Solaris [10, 18]. Mandatory locks are shared for reads but exclusive for writes and must be acquired by all processes that read from or write to a file. Valor adds these additional features: (1) a locking permission bit for owner, group, and all (LPerm), (2) a lock policy system call for specifying how locks are distributed upon `exit`, and (3) the ability to lock a directory (and the requirement to acquire this lock for directory operations). System calls performed by non-transactional processes that write to a file, inode, or directory object acquire the appropriate lock before performing the operation and then release the lock upon returning from the call. Non-transactional system calls are consequently two-phase

with respect to exclusive locks and well-formed with respect to writes. Thus Valor provides degree 1 isolation. In this environment, then, by the degrees of isolation theorem [7], transactional processes that obey higher degrees of isolation can have transactions with repeatable reads (degree 3) and no lost updates (degree 2).

Valor supports inter-process transactions by implementing inter-process locking. Processes may specify (1) if their locks can be recursively acquired by their children, and (2) if a child's locks are released or instead given to its parent when the child exits. These specifications are propagated to the Extended Mandatory Locking system with the `Lock Policy` system call.

Valor prevents misuse of locks by allowing a process to acquire a lock only under one of two circumstances: (1) if the process has permission to acquire a lock on the file according to the LPerm of the file, or (2) if the process has read access or write access, depending on the type of the lock. Only the owner of a file can change the LPerm, but changes to the LPerm take effect regardless of transactions' isolation semantics. Deadlock is prevented using a deadlock-detection algorithm. If a lock would create a circular dependency, then an error is returned. Transaction-aware processes can then recover gracefully. Transaction-oblivious processes should check the status of the failed system call and return an error so that they can be aborted. This works well in practice. We have successfully booted, used, and shutdown a previous version of the Valor system with extended mandatory locking and the standard legacy programs. A related issue is the locking of frequently accessed file-system objects or pages. The default Valor behavior is to provide degree 1 isolation, which prevents another transaction from accessing the page while another transaction is writing to it. For transaction-oblivious processes, because each individual system call is treated as a transaction, these locks are short lived. For transaction-aware processes, an appropriate level of isolation can be chosen (e.g., degree 2—no lost updates) to maximize concurrency and still provide the required isolation properties.

3.3 Application Interception

Valor supports applications that are aware of transactions but need to invoke subprocesses that are not transaction-aware within a transaction. Such a subprocess is wrapped in a transaction that begins when it first performs a file operation and ends when it exits. This is useful for a transactional process that forks subprocesses (e.g., `grep`) to do work within a transaction. During system calls, Valor checks a flag in the process to determine whether to behave transactionally or not. In particular, when a process is `forked`, it can specify if its child is transaction-oblivious. If so, the child has its

Transaction ID set to that of the parent and its in-flight state set to Oblivious. When the process performs any system call that constitutes a read or a write on a file, inode, or directory object, the in-flight state is checked, and an appropriate `Log Append` call is made with the Transaction ID of the process.

4 Evaluation

Valor provides atomicity, isolation, and durability, but these properties come at a cost: writes between the log device and other disks must be ordered, transactional writes incur additional reads and writes, and in-memory data structures must be maintained. Additionally, Valor is designed to provide these features while only requiring minor changes to a standard kernel's design. In this section we evaluate the performance of our Valor design and also compare it to stasis and BDB. Section 4.1 describes our experimental setup. Section 4.2 analyzes a benchmark based on an idealized ARIES transaction logger to derive a lower bound on overhead. Section 4.3 evaluates Valor's performance for a serial file overwrite. Section 4.4 evaluate Valor's transaction throughput. Section 4.5 analyzes Valor's concurrent performance. Finally, Section 4.6 measure Valor's recovery time. All benchmarks test scalability.

4.1 Experimental Setup

We used four identical machines, each with a 2.8GHz Xeon CPU and 1GB of RAM for benchmarking. Each machine was equipped with six Maxtor DiamondMax 10 7,200 RPM 250GB SATA disks and ran CentOS 5.2 with the latest updates as of September 6, 2008. To ensure a cold cache and an equivalent block layout on disk, we ran each iteration of the relevant benchmark on a newly formatted file system with as few services running as possible. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student's-*t* distribution. In each case, unless otherwise noted, the half widths of the intervals were less than 5% of the mean. Wait time is elapsed time less system and user time and mostly measures time performing I/O, though it can also be affected by process scheduling. We benchmarked Valor on the modified Valor kernel, and all other systems on a stable unmodified 2.6.25 Linux kernel.

Comparison to Berkeley DB and Stasis. The most similar technologies to Valor are Stasis and Berkeley DB (BDB): two user level logging libraries that provide transactional semantics on top of a page store for transactions with atomicity and isolation and with or without durability. Valor, Stasis, and BDB were all configured to store their logs on a separate disk from their data, a standard configuration for systems with more

than one disk [7]. The logs used by Valor, Stasis, and BDB were set to 128MiB. Since Valor prioritizes non-durable transactions, we configured Stasis and BDB to also use non-durable transactions. This configuration required modifying the source code of Stasis to open its log without `O_SYNC` mode. Similarly, we configured BDB's environment with `DB_TXN_WRITE_NOSYNC`. The `ext3` file system performs writes asynchronously by default. For file-system workloads it is important to be able to perform efficient asynchronous serial writes, so non-durable transactions performing asynchronous serial writes were the focus during our benchmarking. BDB indexed each page in the file by its page offset and file ID (an identifier similar to an inode number). We used the B-Tree access method as this is the suitable choice for a large file system [44].

4.2 Mock ARIES Lower Bound

Figure 4 compares Valor's performance against a mock ARIES transaction system to see how close Valor comes to the ideal performance for its chosen logging system. We configured a separate logging block device with `ext2`, in order to avoid overhead from unnecessary journaling in the file system. We configured the data block device with `ext3`, since journaled file systems are in common use for file storage. We benchmarked a 2GiB file overwrite under three mock systems. MT-ow-noread performed the overwrite by writing zeros to the `ext2` device to simulate logging, and then writing zeros to the `ext3` device to simulate write back of dirty pages. MT-ow differs from MT-ow-noread in that it copies a pre-existing 2GiB data file to the log to simulate time spent reading in the before image. MT-ow-finite differs from the other mock systems in that it uses a 128MiB log, forcing it to break its operation into a series of 128MiB copies into the log file and writes to the data file. A transaction manager based on the ARIES design must do at least as much I/O as MT-ow-finite. Valor's overhead on top of MT-ow-finite is 35%. Stasis's is 104%. The cost of MT-ow reading the before images as measured by the overhead of MT-ow on MT-ow-noread is only 2%. The cost of MT-ow-finite restricting itself to a finite log is 16% due to required additional seeking. Stasis's overhead is more than Valor's overhead due to maintaining a redundant page cache in user space.

4.3 Serial Overwrite

In this benchmark we measure the time it takes for a process to transactionally overwrite an existing file. File transfers are an important workload for a file system. See Figure 5. Providing transactional protection to large file overwrites demonstrates Valor's ability to scale with larger workloads and handle typical file system operations. Since there is data on the disk already, all sys-

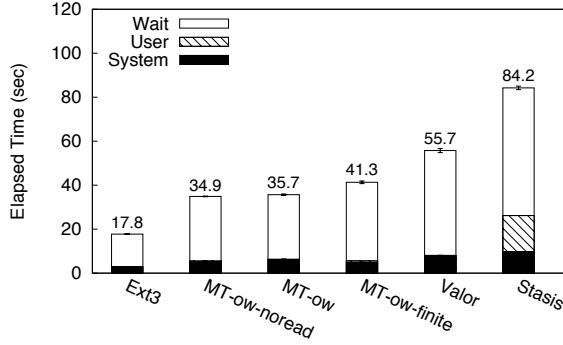


Figure 4: Valor and Stasis's performance relative to Mock ARIES Lower Bound

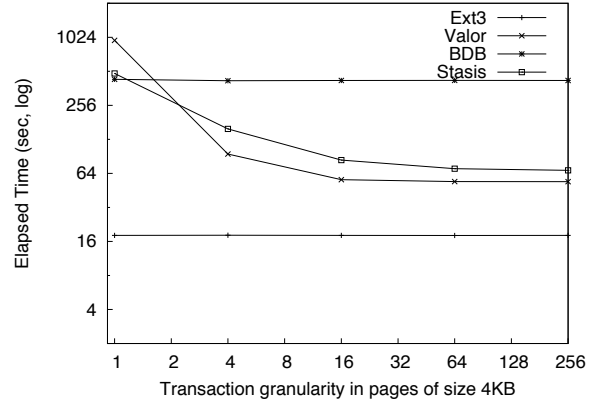


Figure 6: Run times for transactions, increasing granularity

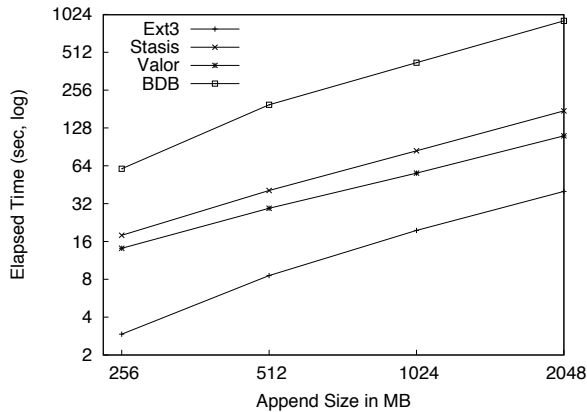


Figure 5: Async serial overwrite of files of varying sizes

tems but `ext3` must log the contents of the page before overwriting it. The transactional systems use a transaction size of 16 pages. The primary observation from these results is that each system scales linearly with respect to the amount of data it must write. Valor runs 2.75 times longer than `ext3`, spending the majority of that overhead writing Log Records to the Log Device. Stasis runs 1.75 times slower than Valor. It spends additional time allocating pages in user space for its own page cache, and doing additional memory copies for its writes to both its log and its store file. For the 512 MiB over write of Valor and Stasis, and the 256 MiB over write of Stasis the half-widths were 11%, 7%, and 23% respectively. The asynchronous nature of the benchmark caused Valor and Stasis' page cache to introduce fluctuations in an otherwise stable serial write. BDB's on-disk B-Tree format, which is very different from Stasis's and Valor's simple page-based layout, makes it difficult to perform well in this I/O intensive workload that has little need for $\log(n)$ B-Tree lookups. Because of this Valor runs 8.22 times the speed of BDB.

4.4 Transaction Granularity

We measured the rate for processing small durable transactions with varying transaction sizes. This benchmark establishes Valor's ability to handle many small transactions, and indicates the overhead of beginning and committing a transaction on a write. We measured BDB, Valor, Stasis, and `ext3`. For `ext3` we simply used the native page size on the disk. See Figure 6. The throughput benchmark is simply the overwrite benchmark from Section 4.3, but we vary the size of the transaction rather than the amount of data to write. We see the typical result that the non-transactional system (`ext3`) is unaffected: transactional systems converge on a constant factor of the non-transactional system's performance as the overhead of beginning and committing a transaction approaches zero. BDB converges on a factor of 23 of `ext3`'s elapsed time, Stasis converges on a factor of 4.2, and Valor converges on a factor of 2.9. It is interesting that Valor has an overhead of 76% with respect to Stasis, and Stasis has an overhead of 25% with respect to BDB for single page transactions. BDB is oriented toward small transactions making updates to a B-Tree, not serial I/O. As the granularity decreases, Stasis and BDB converge to less efficient constant factors of the non-transactional `ext3`'s performance than what Valor converges to. This would imply that Valor's overhead for Log Append is lower than Stasis's since Valor operates from within the kernel and eliminates the need for a redundant page cache. For one page transactions BDB has already converged to a constant factor of `ext3`'s performance starting at 1-page transactions: for transactions less than one page in size, BDB began to perform worse.

4.5 Concurrent Writers

Concurrency is an important measure of how a file interface can handle seeking and less memory while writing. One application of Valor would be to grant atomicity to package managers which may unpack large packages in

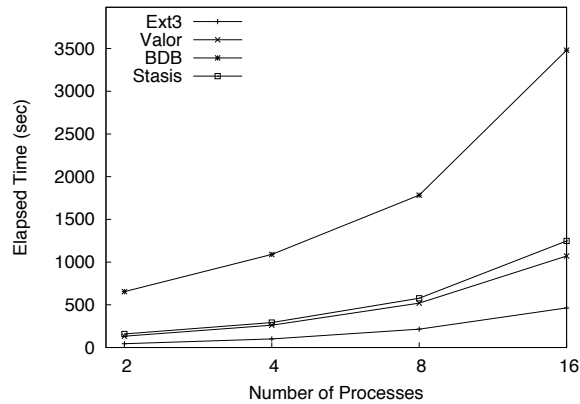


Figure 7: Execution times for multiple concurrent processes accessing different files

parallel. To measure concurrency we ran varying numbers of processes that would each serially overwrite an independent file concurrently. Each process wrote 1GiB of data to its own file, and we ran the benchmark with 2, 4, 8, and 16 processes running concurrently. Figure 7 illustrates the results of our benchmark. For low numbers of processes (2, 4, and 8) BDB had half-widths of 35%, 6%, and %5 because of the high variance introduced by BDB's user space page cache. Stasis and BDB run at 2.7 and 7.5 times the elapsed time of `ext3`. For the 2, 4, 8, and 16 process cases, Valor's elapsed time is 3.0, 2.6, 2.4, and 2.3 times that of `ext3`. What is notable is that these times converge on lower factors of `ext3` for high numbers of concurrent writers. The transactional systems must perform a serial write to a log followed by a random seek and a write for each process. BDB and Stasis must maintain their page caches, and BDB must maintain B-Tree structures on disk and in memory. For small numbers of processes, the additional I/O of writing to Valor's log widens the gap between transactional systems and `ext3`, but as the number of processes and therefore the number of files being written to at once increases, the rate of seeks overtakes the cost of an extra log serial write for each data write, and maintenance of on-disk or in-memory structures for BDB and Stasis.

4.6 Recovery

One of the main goals of a journaling file system is to avoid a lengthy `fsck` on boot [11]. Therefore it is important to ensure Valor can recover from a crash in a finite amount of time with respect to the disk. Valor's ability to recover a file after a crash is based on its logging an equivalent amount of data during operation. The amount of total data that Valor must recover cannot exceed the length of Valor's log, which was 128 MiB in all our benchmarks. Valor's recovery process consists of: (1) reading a page from the log, (2) reading the original

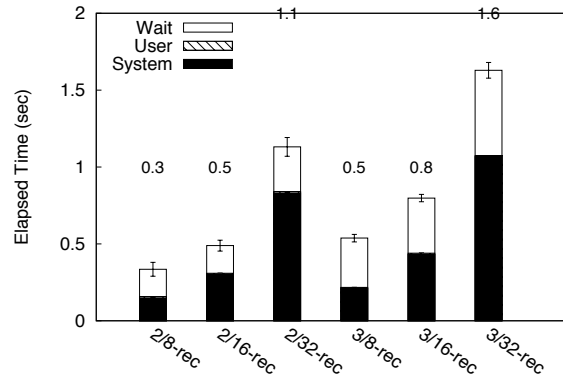


Figure 8: Time spent recovering from a crash for varying amounts of uncommitted data and varying number of processes

page on disk, (3) determining whether to roll forward or back, and (4) writing to the original page if necessary. To see how long Valor took to recover for a typical amount of uncommitted data, we tested the recovery of 8MiB, 16MiB and 32MiB of uncommitted data. In the first trial, two processes were appending to separate files when they crashed, and their writes had to be rolled back by recovery. In the second trial, three processes were appending to separate files. Process crash was simulated by simply calling `exit(2)` and not committing the transaction. Valor first reads the Record Map to reconstitute the in-memory state at the time of crash, then plays each record forward or back in reverse Log Sequence Number (LSN) order. Figure 8 illustrates our recovery results. Label 2/8-rec in the figure shows elapsed time taken by recovery to recover 8MiB of data in the case of 2 process crash. We see that although the amount of time spent recovering is proportional to the amount of uncommitted data for both the 2 and 3 process case, that recovering 3 processes takes more time than for 2 because of additional seeking back and forth between pages on disk associated with log records for 3 uncommitted transactions instead of 2. 2/32-rec is 2.31 times slower than 2/16-rec and 2/16-rec is 1.46 times slower than 2/8-rec due to varying size of recoverable data. Similarly, 3/32-rec is 2.04 times slower than 3/16-rec and 3/16-rec is 1.5 times slower than 3/8-rec. Keeping the amount of recoverable data same we see that 3 processes have 44%, 63%, and 60% overhead compared to 2 process with recoverable data of 8MiB, 16MiB, 32MiB, respectively. In the worst case, Valor recovery can become a random read of 128MiB of log data, followed by another random read of 128MiB of on-disk data, and finally 128MiB of random writes to roll back on-disk data.

Valor does no logging for read-only transactions (e.g., `getdents`, `read`) because they do not modify the file system. Valor only acquires a read lock on the pages be-

ing read, and, because it calls directly down into the file system to service the read request, there is no overhead.

Systems which use an additional layer of software to translate file system operations into database operations and back again introduce additional overhead. This is why Valor achieves good performance with respect to other database-based user level file system implementations that provide transactional semantics. These alternative APIs can perform well in practice, but only if applications use their interface, and constrain their workloads to reads and writes that perform well in a standard database rather than a file system. Our system does not have these restrictions.

5 Conclusions

Applications can benefit greatly from having a POSIX-compliant transactional API that minimizes the number of modifications needed to applications. Such applications can become smaller, faster, more reliable, and more secure—as we have demonstrated in this and prior work. However, adding transaction support to existing OSs is hard to achieve simply and efficiently, as we had explored ourselves in several prototypes.

This paper has several contributions. First, we describe two older prototypes and designs for file-based transactions: (1) KBDBFS which attempted to port a standalone BDB library and add file system support into the Linux kernel—adding over 150,000 complex lines-of-code to the kernel, duplicating much effort; (2) Amino, which moved all that functionality to user level, making it simpler, but incurring high overheads.

The second and primary contribution of this paper is our design of Valor, which was informed by our previous attempts. Valor runs in the kernel cooperating with the kernel's page cache, and runs more efficiently: Valor's performance comes close to the theoretical lower bound for a log-based transaction manager, and scales much better than Amino, BDB, and Stasis.

Unlike KBDBFS, however, Valor integrates seamlessly with the Linux kernel, by utilizing its existing facilities. Valor required less than 100 LoC changes to `pdflush` and another 300 LoC to simply wrap system calls; the rest of Valor is a standalone kernel module which adds less than 4,000 LoC to the stackable file system template Valor was based on.

Future Work. One of our eventual goals is to explore the use of Log Structured Merge Trees [25] to optimize our general purpose log and provide faster name lookups (e.g. decreasing the elapsed time of `find`).

Another interesting research direction is to use NFSv4's compound calls to implement network-based file transactions [38]. This may require semantic change to NFSv4 so as to not allow partial success of some op-

erations within a compound, and to allow the NFS server to perform atomic updates to its back-end storage.

Finally we intend to further investigate the ramifications of weakening `fsync` semantics in light of current trends in hard drive write cache design. We want to explore the possibility of extending asynchronous barrier writes based on native command queueing to the user level layer so that systems which use atomicity mechanisms across multiple devices (e.g., via a logical volume manager or multiple mounts) can retain atomicity. We believe we could avoid hard drive cache flushes [23] using tagged I/O support for SATA drives and export this write ordering primitive to layers higher than the block device and file system implementation. We also are interested in analyzing the probability of failure when using varying semantics for `fsync` as well as analyzing the associated performance trade-offs.

Acknowledgments. We thank the anonymous reviewers and our shepherd, Ohad Rodeh, for their helpful comments. Special thanks go to Russel Sears, Margo Seltzer, Vasily Tarasov, and Chaitanya Yalamanchili for their help evaluating this evolving project. This work was made possible in part thanks to an NSF award CNS-0614784.

References

- [1] D. P. Bovet and M. Cesati. *Understanding the LINUX KERNEL*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2005.
- [2] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: A User-level Versioning File System for Linux. In *Proc. of the Annual USENIX Technical Conf., FREENIX Track*, pp. 19–28, Boston, MA, Jun. 2004.
- [3] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 3rd edition, 2000.
- [4] E. Gal and S. Toledo. A transactional flash file system for microcontrollers. In *Proc. of the Annual USENIX Technical Conf.*, pp. 89–104, Anaheim, CA, Apr. 2005.
- [5] N. H. Gehani, H. V. Jagadish, and W. D. Roome. OdeFS: A File System Interface to an Object-Oriented Database. In *Proc. of the 20th International Conf. on Very Large Databases*, pp. 249–260, Santiago, Chile, Sept. 1994. Springer-Verlag Heidelberg.
- [6] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, 1988.
- [7] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, San Mateo, CA, 1993.
- [8] M. Haardt and M. Coleman. *ptrace(2)*. Linux Programmer's Manual, Section 2, Nov. 1999.
- [9] M. Haardt and M. Coleman. *fsync(2)*. Linux Programmer's Manual, Section 2, 2001.
- [10] M. Haardt and M. Coleman. *fcntl(2)*. Linux Programmer's Manual, Section 2, 2005.
- [11] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. of the 11th ACM Symposium on Operating Systems Principles*, pp. 155–162, Austin, TX, Oct. 1987.

- [12] J. S. Heidemann and G. J. Popek. Performance of cache coherence in stackable filing. In *Proc. of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 3–6, Copper Mountain Resort, CO, Dec. 1995.
- [13] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proc. of the USENIX Winter Technical Conf.*, pp. 235–245, San Francisco, CA, Jan. 1994.
- [14] J. Katcher. PostMark: A new filesystem benchmark. Technical Report TR3022, Network Appliance, 1997. www.netapp.com/tech_library/3022.html.
- [15] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proc. of the Summer USENIX Technical Conf.*, pp. 238–247, Atlanta, GA, Jun. 1986.
- [16] J. MacDonald, H. Reiser, and A. Zarochentcev. Reiser4 transaction design document. www.namesys.com/txn-doc.html, Apr. 2002.
- [17] D. Mazières. A toolkit for user-level file systems. In *Proc. of the Annual USENIX Technical Conf.*, pp. 261–274, Boston, MA, Jun. 2001.
- [18] R. McDougall and J. Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*. Prentice Hall, Upper Saddle River, New Jersey, 2006.
- [19] Microsoft Corporation. Microsoft MSDN WinFS Documentation. <http://msdn.microsoft.com/data/winfs/>, Oct. 2004.
- [20] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.
- [21] N. Murphy, M. Tonkelowitz, and M. Vernal. The Design and Implementation of the Database File System. www.eecs.harvard.edu/~veral/learn/cs261r/index.shtml, Jan. 2002.
- [22] MySQL AB. MySQL: The World’s Most Popular Open Source Database. www.mysql.org, Jul. 2005.
- [23] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, pp. 1–14, Seattle, WA, Nov. 2006.
- [24] M. A. Olson. The Design and Implementation of the Inversion File System. In *Proc. of the Winter 1993 USENIX Technical Conf.*, pp. 205–217, San Diego, CA, Jan. 1993. USENIX.
- [25] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.
- [26] Oracle Corporation. Oracle Internet File System Archive Documentation. http://otn.oracle.com/documentation/ifs_arch.html, Oct. 2000.
- [27] B. Pawlowski, D. Noveck, D. Robinson, and R. Thurlow. The nfs version 4 protocol. In *Proc. of the 2nd International System Administration and Networking Conf.*, page 94, 2000.
- [28] Calton Pu, Jim Johnson, Rogério de Lemos, Andreas Reuter, David Taylor, and Irfan Zakiuddin. 06121 report: Break out session on guaranteed execution. In *Atomicity: A Unifying Concept in Computer Science*, 2006.
- [29] Calton Pu and Jinpeng Wei. A methodical defense against toctou attacks: The edgi approach. In *Proc. of the International Symposium on Secure Software Engineering (ISSSE’06)*, pp. 399–409, Mar. 2006.
- [30] V. K. Reddy and D. Janakiram. Cohesion Analysis in Linux Kernel. *apsec*, 0:461–466, 2006.
- [31] H. Reiser. ReiserFS. www.namesys.com/, Oct. 2004.
- [32] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.
- [33] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pp. 239–253, Pacific Grove, CA, Oct. 1991.
- [34] R. Sears and E. Brewer. Stasis: Flexible Transactional Storage. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, Nov. 2006.
- [35] M. Seltzer and M. Stonebraker. Transaction Support in Read Optimized and Write Optimized File Systems. In *Proc. of the Sixteenth International Conf. on Very Large Databases*, pp. 174–185, Brisbane, Australia, Aug. 1990. Morgan Kaufmann.
- [36] M. I. Seltzer. Transaction support in a log-structured file system. In *Proc. of the Ninth International Conf. on Data Engineering*, pp. 503–510, Vienna, Austria, Apr. 1993.
- [37] M. I. Seltzer. Transaction Support in a Log-Structured File System. In *Proc. of the Ninth International Conf. on Data Engineering*, pp. 503–510, Vienna, Austria, Apr. 1993.
- [38] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. Technical Report RFC 3530, Network Working Group, Apr. 2003.
- [39] Sleepycat Software, Inc. *Berkeley DB Reference Guide*, 4.3.27 edition, Dec. 2004. www.oracle.com/technology/documentation/berkeley-db/db/api_c/frame.html.
- [40] Sun Microsystems, Inc. Solaris ZFS file storage solution. *Solaris 10 Data Sheets*, 2004. www.sun.com/software/solaris/ds/zfs.jsp.
- [41] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proc. of the Annual USENIX Technical Conf.*, pp. 1–14, San Diego, CA, Jan. 1996.
- [42] Stephen Tweedie. Ext3, journaling filesystem. In *Ottawa Linux Symposium*, Jul. 2000. <http://olstrans.sf.net/release/OLS2000-ext3/OLS2000-ext3.html>.
- [43] S. Verma. Transactional NTFS (TxF). <http://msdn2.microsoft.com/en-us/library/aa365456.aspx>, 2006.
- [44] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID Semantics to the File System. *ACM Transactions on Storage (TOS)*, 3(2):1–42, Jun. 2007.
- [45] Peter Zaitsev. True fsync in linux (on ide). Technical report, MySQL AB, Senior Support Engineer, Mar. 2004. lkml.org/lkml/2004/3/17/188.

Understanding Customer Problem Troubleshooting from Storage System Logs

Weihang Jiang[†], Chongfeng Hu[†], Shankar Pasupathy[‡], Arkady Kanevsky[‡], Zhenmin Li[§], Yuanyuan Zhou[†]

[†] University of Illinois

{wjiang3,chu7,yyzhou}@cs.uiuc.edu

[‡]NetApp, Inc.

{shankarp,arkady}@netapp.com

[§] Pattern Insight, Inc.

zhenmin.li@patterninsight.com

Abstract

Customer problem troubleshooting has been a critically important issue for both customers and system providers. This paper makes two major contributions to better understand this topic.

First, it provides one of the first characteristic studies of customer problem troubleshooting using a large set (636,108) of real world customer cases reported from 100,000 commercially deployed storage systems in the last two years. We study the characteristics of customer problem troubleshooting from various dimensions as well as correlation among them. Our results show that while some failures are either benign, or resolved automatically, many others can take hours or days of manual diagnosis to fix. For modern storage systems, hardware failures and misconfigurations dominate customer cases, but software failures take longer time to resolve. Interestingly, a relatively significant percentage of cases are because customers lack sufficient knowledge about the system. We observe that customer problems with attached system logs are invariably resolved much faster than those without logs.

Second, we evaluate the potential of using storage system logs to resolve these problems. Our analysis shows that a failure message alone is a poor indicator of root cause, and that combining failure messages with multiple log events can improve low-level root cause prediction by a factor of three. We then discuss the challenges in log analysis and possible solutions.

1 Introduction

1.1 Motivation

There has been a lot of effort, both academic and commercial [12, 22, 29, 35, 36, 46], put into building robust systems over the past two decades. Despite this, problems always occur at customer sites. Customers usually report such problems to system vendors who are then responsible for diagnosing and fixing the problems. Rapid resolution of customer problems is critical for two reasons. First, failures in the field result in costly downtime

for customers. Second, these problems can be very expensive for system vendors in terms of customer support personnel costs.

A recent study indicates that problem diagnosis related activity is 36–43% of TCO (total cost of ownership) in terms of support costs [17]. Additionally, downtime can cost a customer 18–35% of TCO [17]. The system vendor pays a price as well. A survey showed that vendors devote more than 8% of total revenue and 15% of total employee costs on technical support for customers [52]. The ideal is to automate problem resolution, which can occur in seconds and essentially costs \$0.

Unfortunately, customer problem troubleshooting is very challenging because modern computing environments consist of multiple pieces of hardware and software that are connected in complex ways. For example, a customer running an application, which uses a database on a storage system, might complain about poor performance, but without sophisticated diagnostic information, it is often difficult to tell if the root cause is due to the application, network switches, database, or storage system. Individual components such as storage systems are themselves composed of many interconnected modules, each of which has its own failure modes. For example, a storage system failure can be caused by disks, physical interconnects, shelves, RAID controllers, etc [4, 5, 27, 47, 28]. Furthermore a large fraction of customer problems tend to be human generated misconfiguration [46] or operator mistakes [43].

In all these cases, there is a **problem symptom** (e.g. system failure) and a **problem root cause** (e.g. disk shelf failure). The goal of customer problem troubleshooting is to rapidly identify the root cause from the problem symptom, and apply the appropriate fix such as a software patch, hardware replacement or configuration correction. In some cases the fix is simply to clear a customer's wrong expectation.

It is standard practice for software and hardware providers today to build-in the capability to record important system events in logs [51, 44]. Despite the

widespread existence of logs, there is limited research on the use of logs to troubleshoot system misbehavior or failures. For IP network systems, some fault localization studies use log events to observe the network link failures, while the core diagnosis algorithms rely on the dependency models describing the relationship between link failures and network component faults [32, 30, 3]. For other systems, such a priori knowledge is usually lacking. Other research using logs deals with intrusion detection and security auditing [1, 19]. In industry, Log-logic [39] and Splunk [25] provide solutions to help mine logs for patterns or specific words. While useful, they do not automate system fault diagnosis.

In this paper, we explore the use of storage system logs to troubleshoot customer problems. We start by characterizing the nature of customer problems, and measuring problem resolution time with and without logs (Sections 2 and 3). We then evaluate the extent to which a problem symptom alone can help narrow the possible cause of the problem (Section 4). Finally, we study the challenges in using logs to accurately obtain problem root cause information (Section 5) and briefly outline some ideas we have for automated log analysis (Section 5.3). We are currently evaluating these ideas in a system we are building for fully automated customer troubleshooting from logs.

1.2 Our Findings

Providing meaningful, quantitative answers to the questions we want to explore is a challenging task since it requires analysis of hundreds or thousands of real world customer cases and system logs. We speculate the lack of availability of such a data set is one of the reasons for the absence of studies in this area.

We had access to three structured databases at NetApp containing a wealth of information about customer cases, relevant system logs, and engineering analysis of the customer problems.

Using this data, our work makes two major contributions. First, it provides one of the first characteristic studies of customer problem troubleshooting using a large set (636,108) of real world cases from 100,000 commercially deployed storage systems produced by NetApp. We study the characteristics of customer problem troubleshooting from various dimensions including distribution of root causes, impact, problem resolution time as well as correlation among them. We evaluate the feasibility and challenges of using logs to resolve customer problems and outline a potential automatic log analysis technique.

We have the following major findings:

(1) Problem troubleshooting is a time-consuming and challenging task. While we observed that 36% of reported problems are benign and automatically resolved,

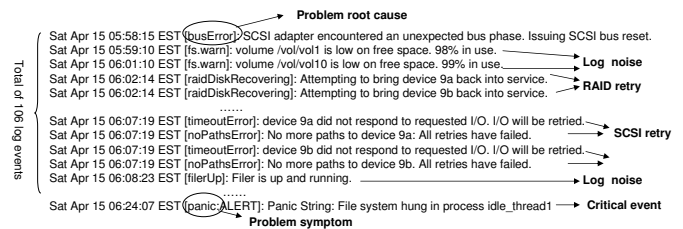


Figure 1. A sample asup message log. The problem symptom is a panic. The root cause is a SCSI bus bridge error. For this root cause, the log has some noise, i.e. events that are not connected with this case.

the remainder required expensive manual intervention that can take a long time.

(2) Hardware failures (40%) and misconfigurations (21%) dominate customer cases. Software bugs account for a small fraction (3%) but can cause significant downtime and take much longer to resolve.

(3) A significant percentage of customer problems (11%) are because customers lack sufficient knowledge about the system, which leads to misconfiguring the operating environment.

(4) More than 87% of problems have low impact because they are handled by built-in failure tolerance mechanisms such as RAID-DP® [16]. While high-impact problems are much fewer, they are much more difficult to troubleshoot due to complex interactions between system modules and the multiple failure modes of these modules.

(5) An important finding is that customer cases with available system log messages invariably have a shorter (16-88%) problem resolution time than cases that don't have logs.

(6) Critical events in logs, which capture the failure symptoms, can help identify the high-level problem category, such as hardware problem, misconfiguration problem, etc. However, on their own, critical events are not enough to identify a more precise problem root cause which is necessary to resolve the customer problem.

(7) Combining critical events with multiple other log events can improve the problem root cause prediction by 3x, except for misconfigurations which tend to have too many noisy, unrelated log events.

(8) Logs are challenging to analyze manually. They contain a lot of log noise, due to messages logged by modules that are not related to the problem. Often log messages are fuzzy as well. This calls for an intelligent log analysis tool to filter out log noise and accurately capture a problem signature.

While we believe that many of our findings can be generalized to other system providers, especially storage system providers, we would still like to caution readers

to take our dataset and evaluation methodology into consideration when interpreting and using our results.

2 Data Sources and Methodology

In this section, we describe how customer cases are created and resolved, and the use of system logs in this process. We also discuss how we select case and log data for analysis.

2.1 The AutoSupport System

The AutoSupport system [33] consists of infrastructure built into the operating system to log system events and to collect and forward these events to a database. While customers can choose if they want to forward these messages to the storage company, in practice most do so since it allows for proactive system monitoring and faster customer support.

Asup messages (autosupport messages) are sent both on a periodic basis and also when critical events occur. Periodic messages contain aggregated information for the week such as average CPU utilization, number of read and write I/Os, ambient temperature, disk space utilization etc. **Critical events** consist of warning messages or failure messages. Warnings, such as a volume being low on space, can be used for proactive resolution. A failure message, such as a system panic or disk failure is diagnosed and fixed, after it is reported.

Every asup message contains a unique id that identifies the system that generated the message, the reason for the message, and any additional data that can help such as previously logged system events, system configuration etc.

2.2 An Example Scenario and Terminology

Figure 1 shows a sample asup message log. At the very end of the log is a *critical event* which is a message showing there was a file system panic that halted the system. Critical events can be either *failure messages* or *warning messages*. The critical event contains a problem symptom, in this case the system panic, which is what the customer observes as the problem.

Notice that every module in the system logs its own messages, and this is part of what makes log analysis very difficult. There is often a lot of log noise, which is what we call log messages that are not relevant to the current problem. As we see in Figure 1, there are over 100 messages in a short span of time, most of which are not relevant to the problem symptom.

In this example, we see that various components below the file system, including, the RAID and the SCSI layer, log their own failure messages. From our analysis, we determined that the *problem root cause* was a SCSI bus failure which is logged 106 events before the problem symptom.

Therefore, manually inspecting these logs can be time consuming. Furthermore, manual inspection requires a good understanding of the interactions between various software layers. In this example, the person resolving the case from logs would need to realize that the SCSI bus failure makes disks unavailable which in turn caused the file system to panic to prevent further writes that could not be safely written to disk.

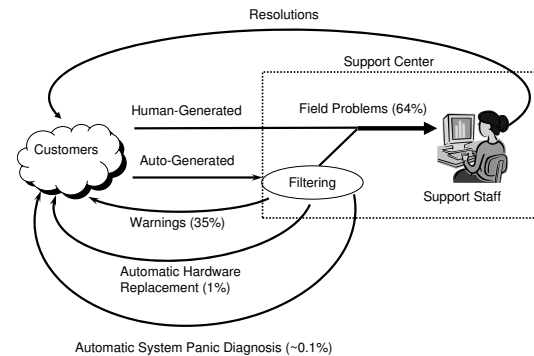


Figure 2. Flowchart of the customer support system

2.3 How Customer Cases are Created

Customer cases are created either automatically or manually. For every asup message that is received by the company, a rule-engine is applied to determine if a customer case should be created in the customer support database. We refer to these cases as *auto-generated cases*. Such cases have a problem symptom, which is the asup failure or warning message that led to the case being opened. For example, a system panic is a symptom that always results in the creation of a customer case.

Human-generated cases are those that are created directly by the customer, either over the phone or by email. These often include performance problems which are difficult to detect and log automatically.

Figure 2 illustrates how customer cases are generated and resolved in the customer support system.

2.4 Customer Case Resolution

Auto-generated customer cases are either manually resolved or automatically resolved. In Figure 2, 35% of customer cases are filtered out by the system since they are warnings that have no immediate customer impact. For 1% of customer cases, for example a disk failure, the resolution is to automatically ship a replacement part. 0.1% of customer cases are system panics that were automatically resolved by comparing the panic message and stack back trace to a knowledge-base and pointing the customer to appropriate fix.

In our study, we focus only on human-generated and auto-generated cases that are manually resolved since

these are the ones that are most expensive both in terms of downtime and financial cost to the customer and the storage system company.

2.5 Data Selection

We now describe how we selected customer case data for analysis in later sections of this paper. There are two primary databases that were used. The first is a *Customer Support Database* that contains details on every customer case that was human-generated or auto-generated. Certain problems that cannot be resolved by customer support staff are escalated to engineering teams, who also record such problems in an *Engineering Case Database*.

We analyzed 636,108 customer cases from the Customer Support Database over the period 1/1/2006 to 1/1/2008. Of these 329,484 customer cases were human-generated and 306,624 customer cases were auto-generated. Overall these represent about 100,000 storage systems.

For each of these 636,108 customer cases, *problem category* and *resolution time* are retrieved from the Customer Support Database. For each of the 306,624 auto-generated customer cases, we also retrieved the critical event that led to the creation of the case. However, the human-generated cases do not have such information.

The goal for resolving any customer case is to determine the problem root cause as soon as possible. Since such information in the Customer Support Database is unstructured, it was difficult to identify problem root cause for solved cases. However, the Engineering Case Database records problem root cause at a fine level. We used 4,769 such cases that were present in both the Customer Support as well as Engineering Case database to analyze problem root cause and its correlation with critical events.

To study the correlation between problem root cause and storage system logs, we retrieve the AutoSupport logs from the AutoSupport Database. Since not all customer systems send AutoSupport logs to the company, among 4,769 customer cases, 4,535 customer cases have corresponding AutoSupport log information.

2.6 Generality of our study

Although our study is based on customer service workflow at NetApp, we believe it is quite representative. As defined in ITIL [57], this customer service workflow represents a typical troubleshooting sequence: a problem case is opened by a call to the help center or by an alert generated by a monitoring system, followed by diagnosis by support staff. A similar process is followed by IBM customer service as described in [24]. Moreover, the comprehensive environment of the storage systems, gives us an opportunity to study a mixture of hardware, software and configuration problems.

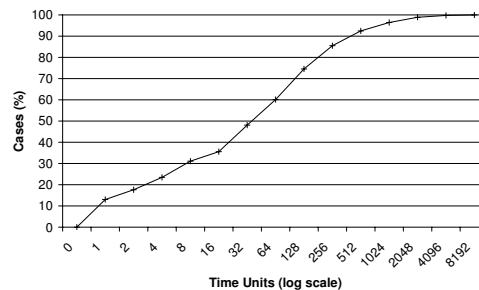


Figure 3. Cumulative Distribution Function (CDF) of resolution time for all customer cases. ¹ There is wide variance in problem resolution time, with some cases taking days to solve.

3 Characteristics of Field Problems

3.1 Problem Resolution Time

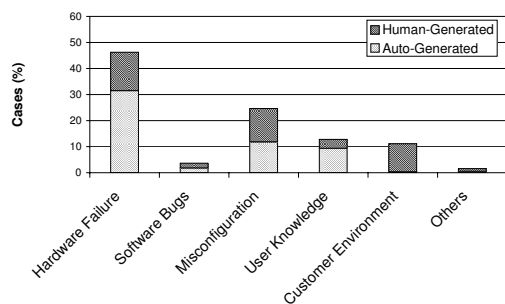
One of the most important metrics of customer support is problem resolution time, which is time spent between when a case is opened and when the resolution or workaround is available for a customer. The distribution of problem resolution times is the key to understanding the complexity of a specific problem or problem class, since it mostly reflects the amount of time spent on troubleshooting problems. It is important to notice that it should not be directly used to calculate MTTR (Mean Time To Recovery), since it does not capture the amount of time to completely solve the problems (e.g., for hardware related problems, it does not include hardware replacement or when it is scheduled to minimize the impact for users).

Figure 3 shows the Cumulative Distribution Function (CDF) of resolution time for all customer cases selected from the Customer Support Database. It is possible for troubleshooting to take many hours. For a small fraction of cases, resolution time can be even longer. Since the x-axis of the figure is logarithmic, the graph shows that doubling the amount of time spent on problem resolution does not double the number of cases resolved. While the Autosupport logging system is an important step in helping troubleshoot problems, this figure makes the case that better tools and techniques are needed to reduce problem resolution time.

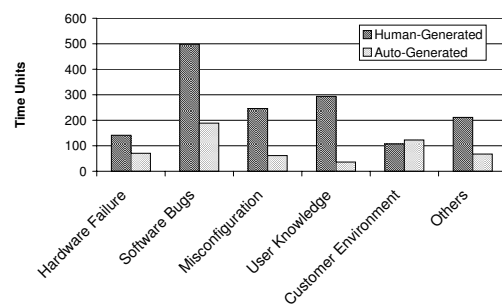
3.2 Problem Root Cause Categories

Analyzing the distribution of problem root causes is useful in understanding where one should spend effort when troubleshooting customer cases or designing more robust systems. While a problem root cause is precise, such as a SCSI bus failure, in this section we lump root causes into categories such as hardware, software, mis-configuration, etc. For all the customer cases, we study

¹We anonymize results to preserve confidentiality and anonymity.



(a) Categorization of Problem Root Causes



(b) Average Resolution Time per Problem Root Cause Category¹

Figure 4. Problem Root Cause Category.

Hardware Failure is related to problems with hardware components, such as disk drive. *Software Bug* is related to storage system software, and *Misconfiguration* is related to system problems caused by errors in configuration. *User Knowledge* is related to technical questions, e.g., explaining why customers were seeing certain system behaviors. *Customer Environment* is related to problems not caused by storage system itself. The figures shows that hardware failures and misconfiguration problems are the major root causes, but software bugs took longer time to resolve.

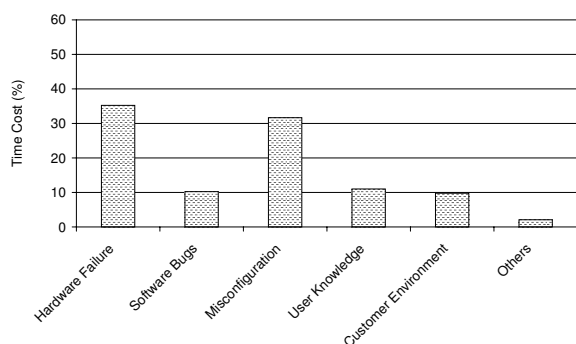


Figure 5. Resolution Time Spent on Problem Root Cause by Category.

Although software problems take longer time to resolve on average, hardware failure and misconfiguration related problems have greater impact on customer experience.

resolution time for each category, relative frequency of cases in each category, and the cost which is the average resolution time multiplied by the number of cases for that category.

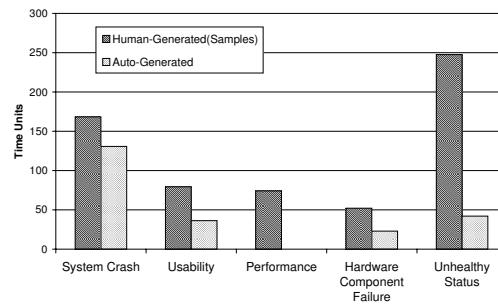
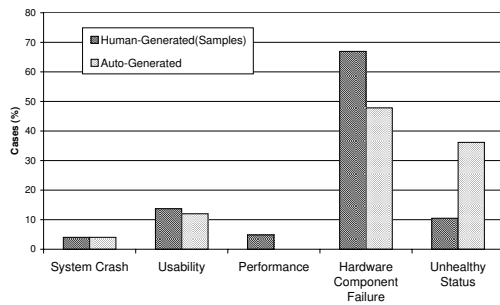
As Figure 4 (a) shows, hardware failures and misconfiguration are the two most frequent problem root cause categories, and contribute 40% and 21% to all customer cases, respectively. Software bugs account for a small fraction (3%) of cases. We speculate that software bugs are not that common since software undergoes rigorous tests before being shipped to customers. Besides tests, there are many techniques [12, 29, 35, 36] that can be applied to find bugs in software. While on average, based on figure 4 (b), software bugs take a longer time to resolve, since their number is so small their overall impact on total time spent on all problem resolutions is not very high, as Figure 5 shows.

It is interesting to observe that a relatively significant percentage of customer problems are because customers lack sufficient knowledge about the system (11%) or customers' own execution environments are incorrect (9%) (e.g. a backup failure caused by a Domain Name System error). These problems can potentially be reduced by providing more system training programs or better configuration checkers.

Figure 4 (b) is our first indication that logs are indeed useful in reducing problem resolution time. Auto-generated customer cases i.e. those with an attached system log and problem symptom in the form of a critical event message, take less time to resolve than human-generated cases. The latter are often poorly defined over the phone or by email. The only instance where this is not true is when the problem relates to the customer's environment, which is difficult to record via an automated system.

3.3 Problem Impact

In the previous subsections, we have treated all problems as equal in their impact on customers. We now consider customer impact for each problem category. To do this, we divide customer cases into 6 categories based on impact ranging from system crash which is the most serious, to low impact unhealthy status. The other categories from higher to lower impact are usability (e.g. inability to access a volume), performance, hardware component failure, and unhealthy status (e.g., instability of the interconnects, low spare disk count). Hardware failures typically have low impact since the storage systems are designed to tolerate multiple disk failures [16], power-supply failures, filer head failures etc. However, until the failed component is replaced, the system operates in degraded mode where the potential for complete system



(a) Distribution of Problems with Different Impact (b) Average Resolution Time of Problem with Different Impact¹
Figure 6. Problem Impact.² From the left to the right, it is in the order of higher impact to lower impact on customer experience. Although the problems with higher impact happen much less frequently compared to the problems with lower impacts, they are usually more complicated to resolve.

failure exists, should its redundant component fail.

Since human-generated customer cases do not have all impact information in structured format, we randomly sampled 200 human-generated cases and manually analyzed them. For auto-generated problems, we include all the cases, and leverage the information in Customer Support Database.

For both human-generated and auto-generated cases, the classification is exclusive: each problem case is classified to one and only one category. The classification is based on how a problem impacts customers' experience. For example, a disk failure that led to a system panic will be classified as an instance of *System Crash*. If it did not lead to system crash (i.e. RAID handled it) it is classified as an instance of *Hardware Component Failure*. It is important to notice that, in our study the *Performance* problems are problem cases that lead to unexpected performance slowdown. Therefore disk failures leading to expected slowdown with RAID reconstruction processes are classified as *Hardware Component Failures*, instead of *Performance* problems.

Figure 6 (a) shows the distribution of problems by impact. One obvious observation is that there are far fewer high-impact problems than low-impact ones. More specifically, *system crash* only contributes about 3%, and *usability* problems contribute about 10%. Low impact problems such as *hardware component failure* and *unhealthy status* contribute about 44% and 20%, respectively.

While high-impact problems are much fewer, as Figure 6 (b) shows, they are more time consuming to troubleshoot. This is due to the complex interaction between system modules. For example, the problem shown in Figure 1 resulted in a system crash. The root cause was an error in the SCSI bus bridge. This started a chain of recovery mechanisms in layers of software, including retries by the RAID layer and SCSI layer. As the result, the time from the root cause to system failure is about a half

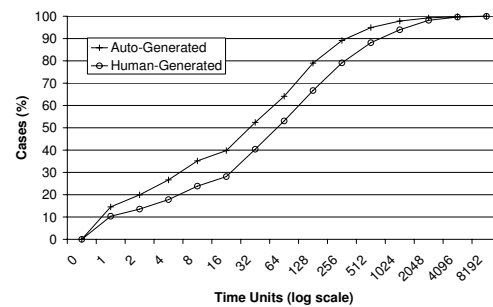


Figure 7. Case Generation Method and Resolution Time.¹ Auto-Generated problems are resolved faster than Human-Generated problems.

hour, and there are more than 100 log events in between the critical event and problem root cause. This makes manual diagnosis of such problems difficult, even when logs are available.

Finally, as we observed in the previous section, auto-generated cases take less time to resolve than human-generated ones.

3.4 Customer case generation method

As we mentioned in Section 2, 51.6% customer cases were human-generated and 48.4% were auto-generated. We now look at how these two methods impact resolution time.

Figure 7 shows that resolution time for auto-generated and human-generated customer cases is similar in distribution: both show huge variance in time. On the other hand, auto-generated cases were solved faster than human-generated ones.

One possible reason why auto-generated cases can be resolved faster than human-generated ones is that auto-generated cases contain valuable information such as

²“System Crash” here means crash of single system, which might not lead to service downtime with a cluster configuration.

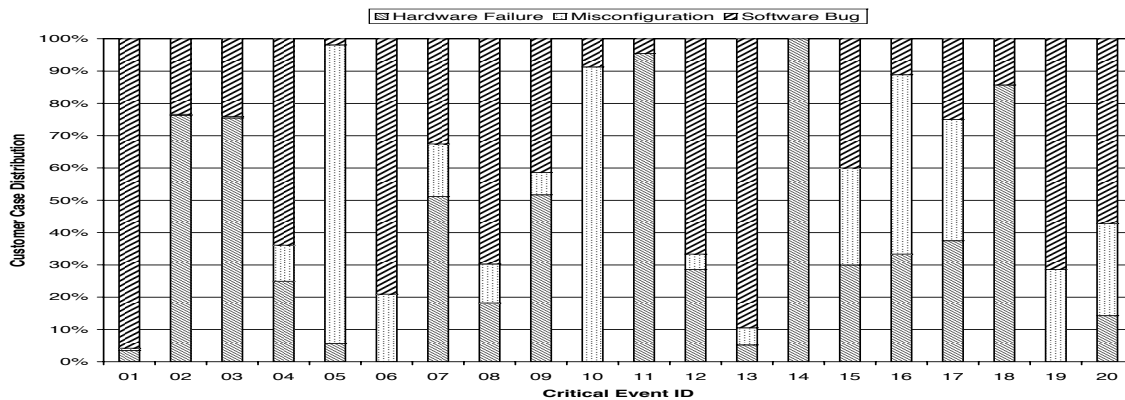


Figure 8. Critical Events can partially help infer high-level problem root causes. The distribution of customer cases across problem root cause categories, for 20 of the most common critical events. 4,769 auto-generated customer cases that contain detailed root cause diagnosis were selected from the Engineering Case Database for this analysis.

critical events, which capture problem symptoms. In addition, information on prior failures or warnings is available in the system's logs.

In comparison, human-generated problems are usually sent with vague descriptions, which vary from one person to another and this information does not have the same rigorous structure as auto-generated ones.

Similar trends have been observed in Figure 4(b). Across all problem root cause categories, auto-generated cases take 16-88% less resolution time than human-generated cases. The only exception is Customer Environment cases, where auto-generated and human-generated cases take similar average resolution time.

4 Can Critical Events Help Infer Root Causes?

Having established that customer cases with attached system logs result in improved problem resolution time, we now ask if critical events in the logs can be directly used to identify problem root cause. To remind the reader, a critical event is a special kind of log message that contains a problem symptom, and triggers the automatic opening of a customer case via the Autosupport system. An example of such an event is a system panic log message.

4.1 High-level Problem Root Causes

We first look at the relationship between critical events and high-level problem root cause categories: hardware failure, software bug, and misconfiguration. We do not present the results for the other two problem root cause categories (user knowledge and customer environment) because they are often human-generated and rarely have a clear critical event in the system log.

Figure 8 shows the distribution of customer cases amongst the three high-level root cause categories for the

Case A

```
Sun Aug 5 08:26:39 CDT [downloadRequest]: newer system software download requested.
Sun Aug 5 08:29:38 CDT [downloadRequestDone]: download complete.
Sun Aug 5 08:34:36 CDT [raidLabelUpgrade]: upgrade RAID labels.
Sun Aug 5 08:34:56 CDT [diskLabelBroken]: device 1 has a broken label.
Sun Aug 5 08:34:56 CDT [diskLabelBroken]: device 2 has a broken label.
...
Sun Aug 5 08:37:42 CDT [raidVolumeFailure: ALERT]: RAID volume 1 has failed.
```

Case B

```
Wed Jan 14 09:41:13 CET [raidDiskInsert]: device 7 inserted.
Wed Jan 14 09:42:57 CET [raidMissingChild]: RAID object 0 only has 1 child, expecting 18.
Wed Jan 14 09:44:05 CET [raidVolumeFailure: ALERT]: RAID volume 2 has failed.
```

Figure 9. Two real-world customer cases with the same critical event: RAID Volume Failure but different root causes. Case A was caused by a software bug: large-capacity disks, which were previously used in degraded-mode (not used in full capacity), were used in full capacity after a software upgrade. However, due to a software bug, disk labels could not be correctly recognized and multiple broken labels led to a *RAID Volume Failure Message*. Case B was caused by misconfiguration: customers mistakenly inserted non-zeroed disk into the system, leading to a *RAID Volume Failure Message*.

20 most frequent critical events. For this experiment, we selected those auto-generated customer cases from the Customer Support Database that were also in the Engineering Case Database, so that we could relate each customer case to its detailed engineering diagnosis.

As seen in Figure 8, for several critical events, there is a dominant high-level problem root cause. For example, 91% of customer cases with critical event 10 (a *Misconfiguration Warning Message*) were obviously diagnosed as misconfiguration problems, and 95% of customer cases with critical event 11 (a *Hardware Failure Warning Message*) were diagnosed as hardware failure

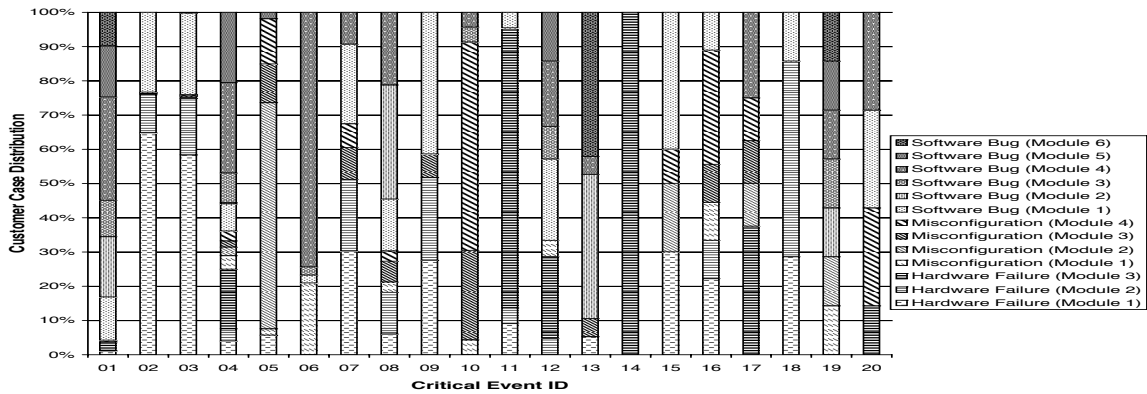


Figure 10. Critical Events cannot infer module-level problem root causes.

Case C

Tue Feb 21 19:00:01 EST [FibreChannelUnstable]: indicates loop stability problem.
 Tue Feb 21 19:27:25 EST [timeoutError]: device 4a did not respond to requested I/O. I/O will be retried.
 Tue Feb 21 19:27:35 EST [timeoutError]: device 4a did not respond to requested I/O. I/O will be retried.
 ...
 Tue Feb 21 19:28:46 EST [noPathsError]: No more paths to device 4a. All retries have failed.
 Tue Feb 21 19:29:03 EST [diskFailure: ALERT]: device 4a has failed.

Case D

Fri May 19 18:38:29 CEST [ioReassignFail]: device 5a sector 140392917 reassign failed.
 Fri May 19 18:38:34 CEST [ioReassignFail]: device 5a sector 140392918 reassign failed.
 Fri May 19 18:38:40 CEST [ioReassignFail]: device 5a sector 140392919 reassign failed.
 Fri May 19 18:39:17 CEST [thresholdMediumError]: device 5a has crossed the medium error threshold.
 Fri May 19 18:39:53 CEST [diskFailure: ALERT]: device 5a has failed.

Figure 11. Two real-world customer cases with the Disk Failure Message. Customer case C was caused by Fibre Channel loop instability and customer case D was caused by disk medium errors.

problems. This is not surprising, since these critical event messages have clear semantic meaning.

However, some critical events cannot be easily categorized to one dominant high-level problem root cause. One example is critical event 07 (a *RAID Volume Failure Message*). Among customer cases with critical event 07, 51% cases were diagnosed as hardware failure related, 16% cases were diagnosed as caused by misconfiguration, and 33% cases were diagnosed as caused by software bugs.

To better understand why there is not always a 1-1 mapping between critical event and root cause category, we pick (Figure 9) two real-world auto-generated customer cases, which were both triggered by the same critical event: *RAID Volume Failure*. As illustrated by the figure, customer case A was caused by a software bug, while customer case B was caused by a misconfiguration (details are explained in the caption).

For a small majority of common critical events (13 out of 20), there is a dominant (> 65%) high-level problem root cause. Therefore, we conclude that critical events can be used to infer the high-level problem root causes.

However, the high-level root cause isn't enough to resolve the customer's problems. One needs to determine the precise root cause. In the next section, we see if critical events at least help us narrow down the root cause to specific storage system modules.

4.2 Module-level Problem Root Causes

A module-level problem root cause defines which module or component³ caused the problem experienced by the customer. Zooming into one particular module is a significant step towards problem resolution. With such knowledge, customer cases can be effectively assigned to the experts who are familiar with that module.

Figure 10 presents the distribution of module-level problem root causes among the customer cases with the same critical event. The same data set was used as for Figure 8. The selected customer cases were diagnosed with 13 different module-level root causes. The figure shows that for only 4 out of 20 messages, there is a dominant (> 65%) module-level problem root cause. Therefore critical events are not indicative of module-level problem root causes.

One explanation is that modules in the storage stack have complex interactions. Multiple code paths can lead to the same failure symptom. An example is critical event 03 (*Disk Failure Message*), which is quite indicative (> 75%) of a hardware failure; however, an error in multiple hardware modules can lead to this message. Figure 11 illustrates two real-world customer cases triggered by *Disk Failure Messages*. As the figure explains, customer case C was actually due to Fibre Channel Loop instability while customer case D was caused by multiple disk medium errors on the same disk.

Since APIs between modules enforce clean separation between caller and callee, modules tend to log "local" state information i.e. what happens within the module. Theoretically a more sophisticated logging infrastructure could store the interactions between modules and generate the critical events that capture "global" system state.

³We will use module to represent both software module and hardware component in the rest of the paper

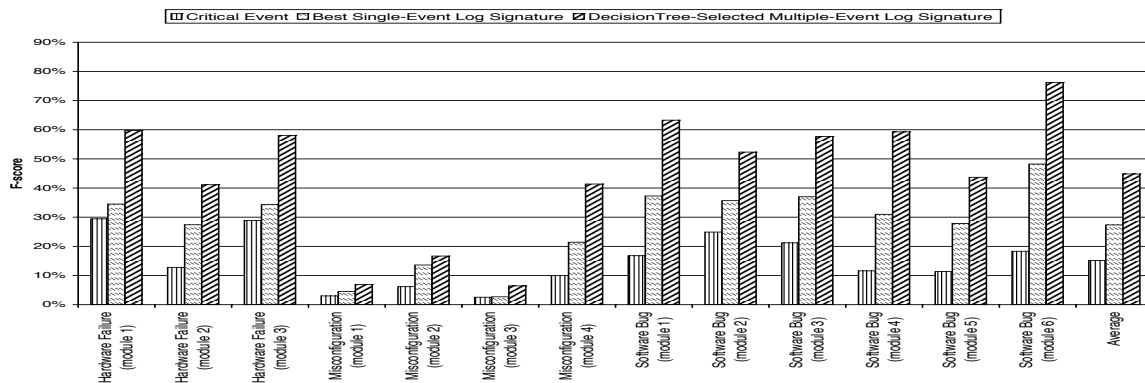


Figure 12. Comparison between three methods of using log events. *F-score* indicates how accurate a prediction can be made on module-level problem root cause using log information. The same set of customer cases are used here as for Figure 8, except customer cases without AutoSupport logs in AutoSupport Database, ending up with 4,535 customer cases.

However, we believe it is impractical to build such logging infrastructure for existing commercial products, due to the complexity of module interaction. Furthermore, such infrastructure would be very hard to maintain as the system evolves and more modules are added. We believe the solution is to combine the critical log event with other log information and in the next section we study the feasibility of doing so.

5 Feasibility of Using Logs for Automating Troubleshooting

As we analyzed in the previous section, critical events alone are not enough for identifying the problem root cause beyond a high level. This conclusion is supported by several real-world customer cases presented in Figure 9 and Figure 11. These customer cases also suggest that log events in addition to the critical events can be quite useful for identifying the problem root causes.

In this section, we investigate the feasibility of using additional information from system logs and answer the following two questions: Does problem root cause determination improve by considering log events beyond critical events? What kind of log events are key to identifying the problem root cause?

5.1 Are additional log events useful ?

To study whether additional log events are useful, we consider three methods of using log event information, and compare how well they can be used as a module-level problem root cause signature. We define a signature as a set of relevant log events that uniquely identify a problem root cause. Such a signature can be used to identify recurring problems and to distinguish one problem from another unrelated one, thereby helping with customer troubleshooting. It is important to note that we are

not designing algorithms to find log signatures, instead we are manually computing log signatures to study how they improve problem root cause determination.

As a baseline, our first method is to only use the problem’s critical event as its signature. For each module-level problem root cause, using a set of manually diagnosed cases as training data, we search for one critical event that can best differentiate customer cases diagnosed with this root cause from other customer cases. More specifically, for each module-level problem root cause, we exhaustively search through all critical events, and calculate their *F-score*, which measures how well the critical event can be used to predict the problem root cause [49]. Then we pick the critical event with the highest *F-score* as the signature for this module-level problem root cause.

The second method is similar to method one. But instead of just looking at critical events to deduce a root cause signature, we search all log events looking for the one log message that best indicated the module-level root cause. If this method can find log signatures with much better *F-score*, it indicates that some log events other than critical events provide more valuable information for identifying problem root cause.

The third method is to use a decision tree [9] to find the best mapping between multiple log events and the problem root cause. The resulting multiple log events can be used as the root cause signature.

For all three methods, we use the same set of customer cases as in Figure 8, except removing customer cases without AutoSupport logs. This gives us 4,535 customer cases. A random selection of 60% of these cases is used as training data, while the remaining 40% are used as testing data.

As Figure 12 shows, for all customer cases, using only

Problem #	Symptom	Cause	# of Key Events	Distance (secs)	Distance (# events)	Fuzziness?
1	Battery Low	Software Bug	2	5.8	1.6	no
2	Shelf Fault	Shelf Intraconnect Defect	3	49.4	3.8	yes
3	System Panic	Broken SCSI Bus Bridge	4	509.2	34.4	no
4	Performance Degradation	FC Loop Defect	2	3652	69.4	no
5(Figure 1)	Power Warning	Incorrect Threshold in Code	2	5	2.4	yes
6(Figure 9A)	RAID Volume Failure	Software Bug	3	196	66.5	no
7(Figure 9B)	RAID Volume Failure	Non-zeroed Disk Insertion	3	80	35	yes
8	RAID Volume Failure	Interconnect Failure	3	290.5	126	yes
9	Shelf Fault	Shelf Module Firmware Bug	4	18285.5	21.5	no
10	Shelf Fault	Power Supply Failure	3	31.5	3.5	no

Table 1. Characteristics of Log Signatures. We manually studied 35 customer cases. These 35 customer cases can be grouped into 10 groups, where each group had the same problem root cause. Based on diagnosis notes from engineers, we were able to identify the key log events, which can differentiate cases in one group cases in another. “# of Key Log Events” is the total number of important log events (including critical events) needed to identify the problem. “Distance” is calculated as the longest distance from a key log event to a critical event for each customer case, averaged across all cases.

critical events as the problem signature is a very poor predictor of root cause. On average, it only achieves an *F-score* of about 0.15. Using the best matched log event, instead of just critical events, can achieve an *F-score* 0.27. By comparison, the average *F-score* achieved by the decision tree method for computing problem signatures is 0.45, which is 3x better than using critical events. Based on these results, we conclude that accurate problem root cause determination requires combining multiple log events rather than a single log event or critical event. This observation matters, since customer support personnel usually focus on the critical event, which can be misleading. Furthermore, as we show in the next section, there is often a lot of noise between key log events making it hard to manually detect problem signatures.

Although we use the decision tree to construct log signatures that are composed of multiple log events, we do not advocate this technique as the solution for utilizing log information. First of all, the accuracy(*F-score*) is still not satisfactory due to log noise, which we discuss later. Moreover, the effectiveness of the decision tree relies on training data. For problem root causes that do not have a large number of diagnosed instances, a decision tree will not provide much help.

5.2 Challenges of using log information

To understand the challenges of using log information and identifying key log events to compute a problem signature, we manually analyzed 35 customer cases sampled from the Engineering Case Database. These customer cases were categorized into 10 groups, such that cases in each group had the same problem root cause.

For these customer cases, we noticed that engineers used several key log events to diagnose the root cause. Table 1 summarizes these cases and characteristics of their key log events.

Based on these 10 groups, we made following major observations:

(1) Logs are noisy.

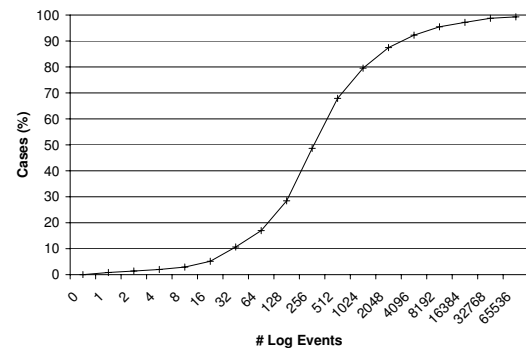


Figure 13. Cumulative Distribution Function (CDF) of number of log events within one hour of critical event. For this figure, we use the same data set as Figure 12. We only count the log events generated and recorded by AutoSupport system within one hour before the critical event, since practically engineers often only exam recent log events for problem diagnosis.

Figure 13 shows the Cumulative Distribution Function (CDF) of the number of log events in AutoSupport logs corresponding to customer cases. As can be seen in the figure, for majority of the customer cases (75%), there are more than 100 log events recorded within an hour before the critical event occurred, and for the top 20% customer cases, more than 1000 log events were recorded.

In comparison, as Table 1 shows, there are usually only 2–4 key log events for a given problem, implying that most log events are just noise for the problem.

(2) Important log events are not easy to locate.

Table 1 shows the distance between key log events and critical events, both in terms of time and the number of log events. For 6 out of 10 problems, at least one key log event is more than 30 log events away from the critical event, which captures the failure point. For all problems, there are always some irrelevant log events in between the key log events and the critical event. In terms of time, the key log events can be minutes or even hours before

the critical event.

(3) The pattern of key log events can be fuzzy.

Sometimes, it is not necessary to have an exact set of key log events for identifying a particular problem. Using problem 7 as an example, it is not necessary to see “raidDiskInsert” log event, depending on how the system administrator added the disk drive. Another example is problem 2. The same shelf intraconnect error can be detected by different modules, and different log messages can be seen for it depending on which module reports the issue.

5.3 Preliminary Prototype for Automatic Log Analysis

Based on the above observations, we designed and implemented a log analysis prototype to improve the customer troubleshooting process. It is important to note, we are still exploring the design space and evaluating the effectiveness of our log analyzer on real world customer cases.

Our analyzer contains two major functions: extracting log signatures and grouping similar logs sequences. As discussed in observation (1), system logs are very “noisy”, containing many log events irrelevant to the problem. We also observed (Table 1) that 2-4 key log events are sufficient to serve as a problem signature.

In order to extract log signatures, our log analyzer automatically ranks log events based on their “importance”. As mentioned in observation (2), important log events are difficult to locate and can be far away from critical events (failure points). To solve this challenge, we apply statistical techniques to infer the dependency between the system states represented by log events. Then we design a heuristic algorithm to estimate the “importance” of a log event based on the following two rules:

(1) Between two dependent log events, the temporally precedent event is usually more important than its successor. If two log events are dependent, the earlier one usually captures the system state that is closer to the beginning of the error propagation process.

(2) The larger dependence “fan-out” a log event has, the more important it is. Our reasoning is that if a log event has a dependence relationship with many other log events and it precedes other log events, it signifies a critical system state.

In this manner, we compute “important” log events for a given problem and rank the top four events which we then use as the problem signature. Even if the signature is not entirely accurate, we believe the process of extracting important events and highlighting those can greatly reduce the time spent by customer support staff in manually analyzing logs.

The second function of our log analyzer is to identify similar log sequences As described in observation (3),

similar log sequences, that represent the same problem root cause, might not have exactly the same set of key log events. Therefore, our log grouping engine clusters logs based on their similarity, by mapping log signatures into a vector space with each log event as a dimension. We then apply unsupervised classification techniques to group similar sequences together based on their relative positions in the vector space [41].

Since we are still exploring the design space and evaluating the effectiveness of our log analysis techniques, the details of the log analyzer are beyond the scope of this paper and remain as our future work.

6 Related Work

6.1 Problem Characteristic Studies

There have been many prior studies that categorize computer system problems and identify root causes such as we have done.

A number of studies show that operator mistakes are one of the major causes of failures. One of the first studies of fault analysis on commercial fault-tolerant systems [21] analyzes Tandem System outages with more than 2000 systems in scope. Gray classifies causes into 5 major categories and 13 sub-categories, and finds that operator error is the largest single cause of failure in deployed Tandem systems. Murphy and Gent examine causes of system crashes in VAX systems between 1985 and 1993, and find that system management caused more than half of the failures, software about 20%, and hardware about 10% [42]. Similarly, the characteristic study by Oppenheimer et al. classifies Internet service failures into component failures and service failures, and further analyzes root causes for each failure type for each Internet service [45]. They also found that operator error is the largest cause of failures in two of the three services, and configuration errors are the largest category of operator errors. While their work focuses on system outages, we are also interested in failures that don’t lead to outages. We classify storage system failures based on symptoms as well as root causes, and further show the correlations between problem root cause, symptom and resolution time.

Ganapathi et al. have developed a categorization framework for Windows registry related problems [20]. Similar to our work, their classification is based on problem manifestation and scope of impact to help understand the problem. Although they have described some causes to problem manifestations, they do not have a clear classification for it. Since our goal is to be able to do problem diagnosis, we study not only the problem symptoms, but also root causes of those symptoms.

Some failure studies are also conducted on storage systems. Jiang et al. conduct a characteristic study of NetApp® storage subsystem failures [27, 28]. They clas-

sify storage subsystem failures into four types, and then study how storage subsystem components can affect storage subsystem reliability.

6.2 Troubleshooting Studies

Since troubleshooting is very time-consuming, quite a few studies have been trying to make it more efficient by automating the process. By studying characteristics of problem tickets in an enterprise IT infrastructure, researchers in IBM T.J. Watson built PDA, a problem diagnosis tool, to help solving problems more efficiently [24]. Banga attempts to automate the diagnosis process of appliance field problems that is usually performed by human experts: system health monitoring and error detection, component sanity checking, and configuration change tracking [6]. Redstone et al. propose a vision of an automated problem diagnosis system by capturing symptoms from users' desktops and matching them against problem database [48].

In order to make this process automated, knowledge about detection and checking rules and logic has to be predefined by human experts. Cohen et al. present a method for extracting signatures from system states to help identify recurrent problems and leverage previous diagnosis efforts [15]. Alternatively, by comparing the target configuration file with the mass of healthy configuration files [55], Wang et al. identified problematic configuration entries that cause Windows® system problems. Similarly, Wang [56] and Lao [34] address misconfiguration problems in Windows systems by building and identifying signatures of normal and abnormal Windows Registry entries. Some studies apply some advanced techniques such as data mining to troubleshooting. For example, PinPoint [14, 13] traces and collects requests, and performs data clustering analysis on them to determine the combinations of components that are likely to be the cause of failures.

It is important to collect system traces for troubleshooting like AutoSupport logging systems. Magpie [7], Flight Data Recorder [54], and the work by Yuan et al. [58] improve system management by using fine-grained system event-tracing mechanisms and analysis. Stack back traces are used by several diagnostic systems, including Dr. Watson [18], Gnome's bug-buddy [11], and IBM diagnosis tool [40].

6.3 Log Analysis

There are two major directions taken by previous researchers to analyze system logs: tupling and dependency extraction.

As a system failure may propagate through multiple system components, multiple log events indicating failure or abnormal status of components can be generated during a short period of time. Based on this observation, several studies try to reduce the complexity of

system logs by grouping successive log events into tuples [8, 10, 23, 26, 37, 38, 53]. For example, Tsao [53], Hansen [23] and Lin [38] applied variants of tupling algorithms on system logs collected from VAX/VMS machines. The tupling algorithms explore the time-space relationship between log events, and cluster temporally related events into tuples, so that the number of logical entities can be significantly reduced. The limitation of tupling algorithms is that log events in a tuple may be unrelated if related log events are interleaving with irrelevant log events. Unfortunately, based on our study on modern system logs, such a limitation is fatal.

Another direction taken by previous studies is to extract dependency between log events. Steinle et al. [50] apply two data mining techniques, aiming at finding the dependency between two events in a log collected from Geneva university hospitals environment. The first technique estimates the distribution of temporal distance between two events, and compares against random distribution. The second technique extracts the correlation between two event types using association statistics. Aguilera et al. [2] apply signal processing techniques to extract dependency between events. The main hypothesis behind this work is that if two events are correlated, one or a few typical temporal gaps between these two events can be found through signal processing. Our study is focused on characteristic study on manually identified key log events, and discusses the challenges and opportunities for applying log analysis. Several observations made in our study using storage system logs are consistent with conclusions made in [31]. Both studies identified that the noisy and redundant log information make log analysis a challenging task and there is great value to extract event correlations for capturing error context and propagation. However, comparing to [31], which made a qualitative study using 2-week distributed system logs, our study looked at 4,769 storage system log files with the corresponding real-world problem diagnosis, carried out a quantitative study on the usefulness of logs, and proposed an automatic log analysis solution.

7 Conclusion

In this paper, we present one of the first studies of the characteristics of customer problem troubleshooting from logs, using a large set of customer support cases from NetApp. Our results show that customer problem troubleshooting is a very time consuming and challenging task, and can benefit from automation to speedup resolution time. We observed that customer problems with attached logs were invariably resolved sooner than those without logs. We show that while a single log event, or critical log event is a poor predictor of problem root cause, combining multiple key log events leads to a 3x improvement in root cause determination. Our results

also show that logs are challenging to analyze manually because they are noisy and that key log events are often separated by hundreds of unrelated log messages. We then outlined our ideas for an automatic log analysis tool that can speed up problem resolution time.

Similar to other characteristic studies, it is impossible to study a handful of different data sets, especially for customer support problems due to the unavailability of such data sets. Even though our data set (which is already very large with 636,108 cases from 100,000 systems) is limited only to NetApp, we believe that this study is an important first-step in quantifying both the usefulness of and challenge in using logs for customer problem troubleshooting. We hope that our study can inspire and motivate characteristic studies about other kinds of systems as well, and motivate the creation of new tools for automated log analysis for customer problem troubleshooting.

References

- [1] C. Abad, J. Taylor, C. Sengul, W. Yurcik, Y. Zhou, and K. Rowe. Log correlation for intrusion detection: A proof of concept. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, page 255, Washington, DC, USA, 2003. IEEE Computer Society.
- [2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of SOSP*, Bolton Landing, NY, Oct. 2003.
- [3] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 13–24, New York, NY, USA, 2007. ACM.
- [4] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. *SIGMETRICS Perform. Eval. Rev.*, 35(1):289–300, 2007.
- [5] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST '08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, San Jose, CA, Feb. 2008.
- [6] G. Banga. Auto-diagnosis of field problems in an appliance operating system. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 24–24, Berkeley, CA, USA, 2000. USENIX Association.
- [7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI '04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.
- [8] A. Bondavalli, S. Chiaradonna, F. D. Giandomenico, and F. Grandoni. Threshold-based mechanisms to discriminate transient from intermittent faults. *IEEE Transactions on Computers*, 49(3):230–245, 2000.
- [9] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Statistics/Probability Series. Wadsworth Publishing Company, Belmont, California, U.S.A., 1984.
- [10] M. F. Buckley and D. P. Siewiorek. A comparative analysis of event tupling schemes. In *FTCS '96: Proceedings of the Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, page 294, Washington, DC, USA, 1996. IEEE Computer Society.
- [11] Bug-buddy. GNOME bug-reporting utility, 2004. http://directory.fsf.org/All_Packages_in_Directory/bugbuddy.html.
- [12] B. Chelf and A. Chou. The next generation of static analysis: Boolean satisfiability and path simulation — a perfect match. In *Coverity White Paper*, 2007.
- [13] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI '04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 23–23, Berkeley, CA, USA, 2004. USENIX Association.
- [14] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.
- [15] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 105–118, New York, NY, USA, 2005. ACM.
- [16] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14, 2004.
- [17] Crimson Consulting Group. The solaris 10 advantage: Understanding the real cost of ownership of red hat enterprise linux. In *Crimson Consulting Group Business White Paper*, 2007.
- [18] Microsoft Corporation. Dr. Watson Overview, 2002. http://www.microsoft.com/TechNet/prodtechnol/winxp/proddocs/drwatson%_overview.asp.
- [19] R. Finlayson and D. Cheriton. Log files: an extended file service exploiting write-once storage. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 139–148, New York, NY, USA, 1987. ACM.
- [20] A. Ganapathi, Y.-M. Wang, N. Lao, and J.-R. Wen. Why pcs are fragile and what we can do about it: A study of windows registry problems. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 561, Washington, DC, USA, 2004. IEEE Computer Society.
- [21] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, 1986.
- [22] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 69–82, New York, NY, USA, 2002. ACM.
- [23] J. P. Hansen. Trend analysis and modeling of uni/multi-processor event logs. In *Master Thesis, Dept. Electrical and Computer Engineering, Carnegie Mellon University*, 1998.
- [24] H. Huang, I. Raymond Jennings, Y. Ruan, R. Sahoo, S. Sahu, and A. Shaikh. PDA: a tool for automated problem determination. In *LISA '07: Proceedings of the 21st conference on 21st Large Installation System Administration Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.
- [25] S. Inc. Why splunk? In *Splunk White Paper*, 2006.
- [26] R. K. Iyer, L. T. Young, and P. V. K. Iyer. Automatic recognition of intermittent failures: An experimental study of field data. *IEEE Trans. Comput.*, 39(4):525–537, 1990.
- [27] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky. Are disks the dominant contributor for storage failures? a comprehensive study of storage subsystem failure characteristics. In *FAST '08: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, 2008.
- [28] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky. Don't blame disks for every storage subsystem failure. In *NetApp Technical Journal*, 2008.
- [29] S. Johnson. Lint, a c program checker, 1978.
- [30] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: a tool for failure diagnosis in ip networks. In *MineNet '05: Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*, pages 173–178, New York, NY, USA, 2005. ACM.
- [31] M. P. Kasick, P. Narasimhan, K. Atkinson, and J. Lepreau. Towards fingerprinting in the emulab dynamic distributed system. In *WORLDS '06: Proceedings of the 3rd conference on USENIX Workshop on Real, Large Distributed Systems*, pages 7–7, Berkeley, CA, USA, 2006. USENIX Association.

- [32] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Ip fault localization via risk modeling. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 57–70, Berkeley, CA, USA, 2005. USENIX Association.
- [33] L. Lancaster and A. Rowe. Measuring real-world data availability. In *LISA '01: Proceedings of the 15th USENIX conference on System administration*, pages 93–100, Berkeley, CA, USA, 2001. USENIX Association.
- [34] N. Lao, J.-R. Wen, W.-Y. Ma, and Y.-M. Wang. Combining high level symptom descriptions and low level state information for configuration fault diagnosis. In *LISA '04: Proceedings of the 18th USENIX conference on System administration*, pages 151–158, Berkeley, CA, USA, 2004. USENIX Association.
- [35] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.
- [36] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 306–315, New York, NY, USA, 2005. ACM.
- [37] Y. Liang, Y. Zhang, H. Xiong, and R. K. Sahoo. An adaptive semantic filter for blue gene/l failure log analysis. In *IPDPS*, pages 1–8, 2007.
- [38] T. T. Y. Lin and D. P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *Reliability, IEEE Transactions on*, 39(4):419–432, 1990.
- [39] I. LogLogic. Logs: Data warehouse style. In *LogLogic White Paper*, 2007.
- [40] G. Lohman, J. Champlin, and P. Sohn. Quickly finding known software problems via automated symptom matching. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 101–110, Washington, DC, USA, 2005. IEEE Computer Society.
- [41] T. Mitchell. Machine learning. McGraw Hill, 1997.
- [42] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International*, 11, 1995.
- [43] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.
- [44] I. Network Appliance. Proactive health management with auto-support. In *NetApp White Paper*, 2007.
- [45] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.
- [46] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery oriented computing (roc): Motivation, definition, techniques,. Technical report, Berkeley, CA, USA, 2002.
- [47] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.
- [48] J. A. Redstone, M. M. Swift, and B. N. Bershad. Using computers to diagnose computer problems. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.
- [49] C. J. V. Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 1979.
- [50] M. Steinle, K. Aberer, S. Girdzijauskas, and C. Lovis. Mapping moving landscapes by mining mountains of logs: novel techniques for dependency model generation. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1093–1102. VLDB Endowment, 2006.
- [51] Linux system logging utilities. Linux Man Page: SYSKLOGD(8).
- [52] The Association of Support Professionals. Technical Support Cost Ratios. 2000.
- [53] M. M. Tsao. Trend analysis and fault prediction. In *PhD Dissertation, Dept. Electrical and Computer Engineering, Carnegie Mellon University*, 1998.
- [54] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight data recorder: monitoring persistent-state interactions to improve systems management. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 117–130, Berkeley, CA, USA, 2006. USENIX Association.
- [55] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 17–17, Berkeley, CA, USA, 2004. USENIX Association.
- [56] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *LISA '03: Proceedings of the 17th USENIX conference on System administration*, pages 159–172, Berkeley, CA, USA, 2003. USENIX Association.
- [57] A. C. Xansa, A. Hanna, C. Rudd, I. Macfarlane, J. Windebank, and S. Rance. An Introductory Overview of ITIL v3. <http://www.itsmfi.org/>, 2007.
- [58] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. *SIGOPS Oper. Syst. Rev.*, 40(4):375–388, 2006.

Trademark Notice: © 2009 NetApp. All rights reserved. Specifications are subject to change without notice. NetApp, the NetApp logo, Go further, faster, and RAIS-DP are trademarks or registered trademark of NetApp, Inc. in the United States and/or other countries. Windows is a registered trademarks of Microsoft Corporation. Linux is a registered trademark of Linus Torvalds.

DIADS: Addressing the “My-Problem-or-Yours” Syndrome with Integrated SAN and Database Diagnosis

Shivnath Babu
Duke University
shivnath@cs.duke.edu

Nedyalko Borisov
Duke University
nedyalko@cs.duke.edu

Sandeep Uttamchandani
IBM Almaden Research Center
sandeepu@us.ibm.com

Ramani Routray
IBM Almaden Research Center
routrayr@us.ibm.com

Aameek Singh
IBM Almaden Research Center
singh@us.ibm.com

Abstract

We present DIADS, an integrated *DI*agnosis tool for Databases and Storage area networks (SANs). Existing diagnosis tools in this domain have a database-only (e.g., [11]) or SAN-only (e.g., [28]) focus. DIADS is a first-of-a-kind framework based on a careful integration of information from the database and SAN subsystems; and is not a simple concatenation of database-only and SAN-only modules. This approach not only increases the accuracy of diagnosis, but also leads to significant improvements in efficiency.

DIADS uses a novel combination of non-intrusive machine learning techniques (e.g., Kernel Density Estimation) and domain knowledge encoded in a new symptoms database design. The machine learning component provides core techniques for problem diagnosis from monitoring data, and domain knowledge acts as checks-and-balances to guide the diagnosis in the right direction. This unique system design enables DIADS to function effectively even in the presence of multiple concurrent problems as well as noisy data prevalent in production environments. We demonstrate the efficacy of our approach through a detailed experimental evaluation of DIADS implemented on a real data center testbed with PostgreSQL databases and an enterprise SAN.

1 Introduction

“The online transaction processing database myOLTP has a 30% slow down in processing time, compared to performance two weeks back.” This is a typical problem ticket a database administrator would create for the SAN administrator to analyze and fix. Unless there is an obvious failure or degradation in the storage hardware or the connectivity fabric, the response to this problem ticket would be: *“The I/O rate for myOLTP tablespace volumes has increased 40%, with increased sequential reads, but the response time is within normal bounds.”* This to-and-fro may continue for a few weeks, often driving SAN administrators to take drastic steps such as migrating the database volumes to a new isolated storage controller or creating a dedicated SAN silo (the inverse

of *consolidation*, explaining in part why large enterprises still continue to have highly under-utilized storage systems). The *myOLTP* problem may be fixed eventually by the database administrator realizing that a change in a table’s properties had made the plan with sequential data scans inefficient; and the I/O path was never an issue.

The above example is a realistic scenario from large enterprises with separate teams of database and SAN administrators, where each team uses tools specific to its own subsystem. With the growing popularity of Software-as-a-Service, this division is even more predominant with application administrators belonging to the customer, while the computing infrastructure is provided and maintained by the service provider administrators. The result is a lack of end-to-end correlated information across the system stack that makes problem diagnosis hard. Problem resolution in such cases may require either *throwing iron* at the problem and re-creating resource silos, or employing highly-paid consultants who understand both databases and SANs to solve the performance problem tickets.

The goal of this paper is to develop an integrated diagnosis tool (called DIADS) that spans the database and the underlying SAN consisting of end-to-end I/O paths with servers, interconnecting network switches and fabric, and storage controllers. The input to DIADS is a problem ticket from the administrator with respect to a degradation in database query performance. The output is a collection of top-K events from the database and SAN that are candidate root causes for the performance degradation. Internally, DIADS analyzes thousands of entries in the performance and event logs of the database and individual SAN devices to shortlist an extremely selective subset for further analysis.

1.1 Challenges in Integrated Diagnosis

Figure 1 shows an integrated database and SAN taxonomy with various logical (e.g., sort and scan operators in a database query plan) and physical components (e.g., server, switch, and storage controller). Diagnosis of problems within the database or SAN subsystem is an

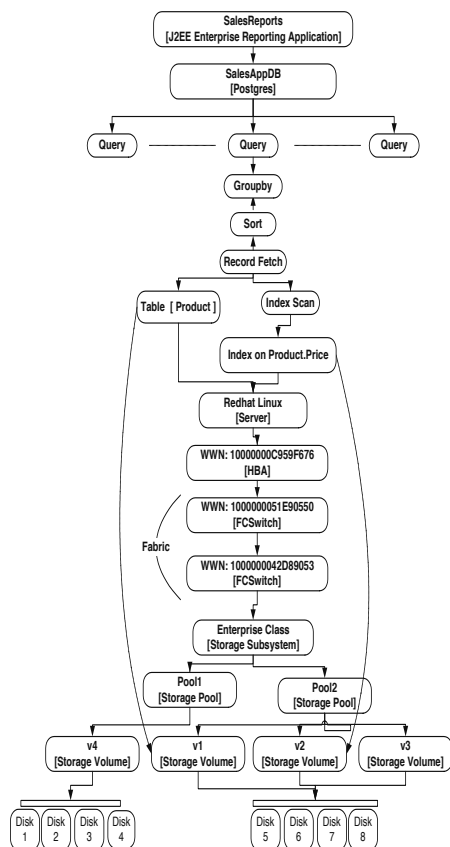


Figure 1: Example database/SAN deployment.

area of ongoing research (described later in Section 2). Integrated diagnosis across multiple subsystems is even more challenging:

- **High-dimensional search space:** Integrated analysis involves a large number of entities and their combinations (see Figure 1). Pure machine learning techniques that aim to find correlations in the raw monitoring data—which may be effective within a single subsystem with few parameters—can be ineffective in the integrated scenario. Additionally, real-world monitoring data has inaccuracies (i.e., the data is *noisy*). The typical source of noise is the large monitoring interval (5 minutes or higher in production environments) which averages out the instantaneous effects of spikes and other bursty behavior.
- **Event cascading and impact analysis:** The cause and effect of a problem may not be contained within a single subsystem (i.e., *event flooding* may result). Analyzing the impact of an event across multiple subsystems is a nontrivial problem.
- **Deficiencies of rule-based approaches:** Existing diagnosis tools for some commercial databases [11] use a rule-based approach where a root-cause taxonomy is created and then complemented with rules

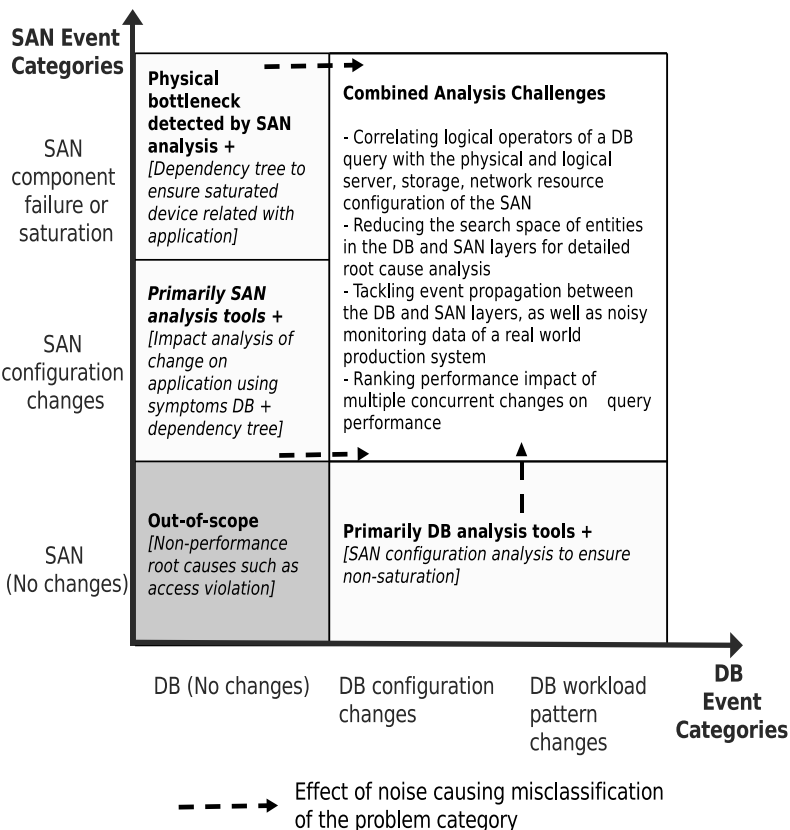


Figure 2: Taxonomy of scenarios for root-cause analysis.

to map observed symptoms to possible root causes. While this approach has the merit of encoding valuable domain knowledge for diagnosis purposes, it may become complex to maintain and customize.

1.2 Contributions

The taxonomy of problem determination scenarios handled by DIADS is shown in Figure 2. The events in the SAN subsystem can be broadly classified into configuration changes (such as allocation of new applications, change in interconnectivity, firmware upgrades, etc.) and component failure or saturation events. Similarly, database events could correspond to changes in the configuration parameters of the database, or a change in the workload characteristics driven by changes in query plans, data properties, etc. The figure represents a matrix of change events, with relatively complex scenarios arising due to combinations of SAN and database events. In real-world systems, the *no change* category is misleading, since there will always be change events recorded in management logs that may not be relevant or may not impact the problem at hand; those events still need to be filtered by the problem determination tool. For completeness, there is another dimension (outside the scope of this paper) representing transient effects, e.g., workload con-

tention causing transient saturation of components.

The key contributions of this paper are:

- A novel workflow for integrated diagnosis that uses an end-to-end canonical representation of database query operations combined with physical and logical entities from the SAN subsystem (referred to as *dependency paths*). DIADS generates these paths by analyzing system configuration data, performance metrics, as well as event data generated by the system or by user-defined triggers.
- The workflow is based on an innovative combination of machine learning, domain knowledge of configuration and events, and impact analysis on query performance. This design enables DIADS to address the integrated diagnosis challenges of high-dimensional space, event propagation, multiple concurrent problems, and noisy data.
- An empirical evaluation of DIADS on a real-world testbed with a PostgreSQL database running on an enterprise-class storage controller. We describe problem injection scenarios including combinations of events in the database and SAN layers, along with a drill-down into intermediate results given by DIADS.

2 Related Work

We give an overview of relevant database (DB), storage, and systems diagnosis work, some of which is complementary and leveraged by our integrated approach.

2.1 Independent DB and Storage Diagnosis

There has been significant prior research in performance diagnosis and problem determination in databases [11, 10, 20] as well as enterprise storage systems [25, 28]. Most of these techniques perform diagnosis in an isolated manner attempting to identify root cause(s) of a performance problem in individual database or storage silos. In contrast, DIADS analyzes and correlates data across the database and storage layers.

DB-only Diagnosis: Oracle's Automatic Database Diagnostic Monitor (ADDM) [10, 11] performs fine-grained monitoring to diagnose database performance problems, and to provide tuning recommendations. A similar system [6] has been proposed for Microsoft SQLServer. (Interested readers can refer to [33] for a survey on database problem diagnosis and self-tuning.) However, these tools are oblivious to the underlying SAN layer. They cannot detect problems in the SAN, or identify storage-level root causes that propagate to the database subsystem.

Storage-only Diagnosis: Similarly, there has been research in problem determination and diagnosis in enterprise storage systems. Genesis [25] uses machine learning to identify abnormalities in SANs. A disk I/O throughput model and statistical techniques to diagnose performance problems in the storage layer are described

in [28]. There has also been work on profiling techniques for local file systems [3, 36] that help collect data useful in identifying performance bottlenecks as well as in developing models of storage behavior [18, 30, 21].

Drawbacks: Independent database and storage analysis can help diagnose problems like deadlocks or disk failures. However, independent analysis may fail to diagnose problems that do not violate conditions in any one layer, rather contribute cumulatively to the overall poor performance. Two additional drawbacks exist. First, it can involve multiple sets of experts and be time consuming. Second, it may lead to spurious corrective actions as problems in one layer will often surface in another layer. For example, slow I/O due to an incorrect storage volume placement may lead a DB administrator to change the query plan. Conversely, a poor query plan that causes a large number of I/Os may lead the storage administrator to provision more storage bandwidth.

Studies measuring the impact of storage systems on database behavior [27, 26] indicate a strong interdependence between the two subsystems, highlighting the importance of an integrated diagnosis tool like DIADS.

2.2 System Diagnosis Techniques

Diagnosing performance problems has been a popular research topic in the general systems community in recent years [32, 8, 9, 35, 4, 19]. Broadly, this work can be split into two categories: (a) systems using machine learning techniques, and (b) systems using domain knowledge. As described later, DIADS uses a novel mix where machine learning provides the core diagnosis techniques while domain knowledge serves as checks-and-balances against spurious correlations.

Diagnosis based on Machine Learning: PeerPressure [32] uses statistical techniques to develop models for a healthy machine, and uses these models to identify *sick* machines. Another proposed method [4] builds models from process performance counters in order to identify anomalous processes that cause computer slowdowns. There is also work on diagnosing problems in multi-tier Web applications using machine learning techniques. For example, modified Bayesian network models [8] and ensembles of probabilistic models [35] that capture system behavior under changing conditions have been used. These approaches treat data collected from each subsystem equally, in effect creating a single table of performance metrics that is input to machine learning modules. In contrast, DIADS adds more structure and semantics to the collected data, e.g., to better understand the impact of database operator performance vs. SAN volume performance. Furthermore, DIADS complements machine learning techniques with domain knowledge.

Diagnosis based on Domain Knowledge: There are also many systems, especially in the DB community, where

domain knowledge is used to create a *symptoms* database that associates performance symptoms with underlying root causes [34, 19, 24, 10, 11]. Commercial vendors like EMC, IBM, and Oracle use symptom databases for problem diagnosis and correction. While these databases are created manually and require expertise and resources to maintain, recent work attempts to partially automate this process [9, 12].

We believe that a suitable mix of machine learning techniques and domain knowledge is required for a diagnosis tool to be useful in practice. Pure machine learning techniques can be misled by spurious correlations in data resulting from noisy data collection or event propagation (where a problem in one component impacts another component). Such effects need to be addressed using appropriate domain knowledge, e.g., component dependencies, symptoms databases, and knowledge of query plan and operator relationships.

It is also important to differentiate DIADS from tracing-based techniques [7, 1] that trace messages through systems end-to-end to identify performance problems and failures. Such tracing techniques require changes in production system deployments and often add significant overhead in day-to-day operations. In contrast, DIADS performs a postmortem analysis of monitored performance data collected at industry-standard intervals to identify performance problems.

Next, we provide an overview of DIADS.

3 Overview of DIADS

Suppose a query Q that a report-generation application issues periodically to the database system shows a slowdown in performance. One approach to track down the cause is to leverage historic monitoring data collected from the entire system. There are several product offerings [13, 15, 16, 17, 31] in the market that collect and persist monitoring data from IT systems.

DIADS uses a commercial storage management server—IBM TotalStorage Productivity Center [17]—that collects monitoring data from multiple layers of the IT stack including databases, servers, and the SAN. The collected data is transformed into a tabular format, and persisted as time-series data in a relational database.

SAN-level data: The collected data includes: (i) configuration of components (both physical and logical), (ii) connectivity among components, (iii) changes in configuration and connectivity information over time, (iv) performance metrics of components, (v) system-generated events (e.g., disk failure, RAID rebuild) and (vi) events generated by user-defined *triggers* [14] (e.g., degradation in volume performance, high workload on storage subsystem).

Database-level data: To execute a query, a database system generates a *plan* that consists of operators selected

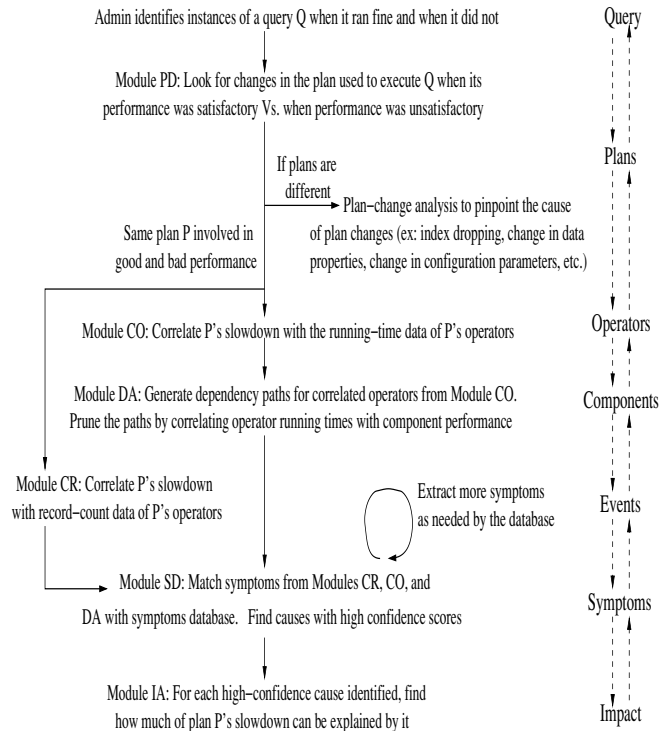


Figure 3: DIADS's diagnosis workflow

from a small, well-defined family of *operators* [14]. Let us consider an example query Q :

```
SELECT  Product.Category, SUM(Product.Sales)
FROM    Product
WHERE   Product.Price > 1000
GROUP BY Product.Category
```

Q asks for the total sales of products, priced above 1000, grouped per category. Figure 1 shows a plan P to execute Q . P consists of four operators: an *Index Scan* of the index on the Price attribute, a *Fetch* to bring matching records from the Product table, a *Sort* to sort these records on Category values, and a *Grouping* to do the grouping and summation. For each execution of P , DIADS collects some monitoring data per operator O . The relevant data includes: O 's start time, stop time, and *record-count* (number of records returned in O 's output).

DIADS's Diagnosis Interface: DIADS presents an interface where an administrator can mark a query as having experienced a slowdown. Furthermore, the administrator either specifies declaratively or marks directly the runs of the query that were *satisfactory* and those that were *unsatisfactory*. For example, runs with running time below 100 seconds are satisfactory, or all runs between 8 AM and 2 PM were satisfactory, and those between 2 PM and 3 PM were unsatisfactory.

Diagnosis Workflow: DIADS then invokes the *workflow* shown in Figure 3 to diagnose the query slowdown based on the monitoring data collected for satisfactory and unsatisfactory runs. By default, the workflow is run in a

batch mode. However, the administrator can choose to run the workflow in an interactive mode where only one module is run at a time. After seeing the results of each module, the administrator can edit the data or results before feeding them to the next module, bypass or reinvoke modules, or stop the workflow. Because of space constraints, we will not discuss the interactive mode further in this paper.

The first module in the workflow, called Module Plan-Diffing (PD), looks for significant changes between the plans used in satisfactory and unsatisfactory runs. If such changes exist, then DIADS tries to pinpoint the cause of the plan changes (which includes, e.g., index addition or dropping, changes in data properties, or changes in configuration parameters used during plan selection). The techniques used in this module contain details specific to databases, so they are covered in a companion paper [5].

The remaining modules are invoked if DIADS finds a plan P that is involved in both satisfactory and unsatisfactory runs of the query. We give a brief overview before diving into the details in Section 4:

- **Module Correlated Operators (CO):** DIADS finds the (nonempty) subset of operators in P whose change in performance correlates with the query slowdown. The operators in this subset are called *correlated operators*.
- **Module Dependency Analysis (DA):** Having identified the correlated operators, DIADS uses a combination of correlation analysis and the configuration and connectivity information collected during monitoring to identify the components in the system whose performance is correlated with the performance of the correlated operators.
- **Module Correlated Record-counts (CR):** Next, DIADS checks whether the change in P 's performance is correlated with the record-counts of P 's operators. If significant correlations exist, then it means that data properties have changed between satisfactory and unsatisfactory runs of P .
- **Module Symptoms Database (SD):** The correlations identified so far are likely *symptoms* of the root cause(s) of query slowdown. Other symptoms may be present in the stream of system-generated events and trigger-generated (user-defined) semantic events. The combination of these symptoms is used to probe a *symptoms database* that maps symptoms to the underlying root cause(s). The symptoms database improves diagnosis accuracy by dealing with the propagation of faults across components as well as missing symptoms, unexpected symptoms (e.g., spurious correlations), and multiple simultaneous problems.
- **Module Impact Analysis (IA):** The symptoms database computes a *confidence score* for each suspected root cause. For each high-confidence root

cause R , DIADS performs impact analysis to answer the following question: if R is really a cause of the query slowdown, then what fraction of the query slowdown can be attributed to R . To the best of our knowledge, DIADS is the first automated diagnosis tool to have an impact-analysis module.

Integrated database/SAN diagnosis: Note that the workflow “drills down” progressively from the level of the query to plans and to operators, and then uses dependency analysis and the symptoms database to further drill down to the level of performance metrics and events in components. Finally, impact analysis is a “roll up” to tie potential root causes back to their impact on the query slowdown. The drill down and roll up are based on a careful integration of information from the database and SAN layers; and is not a simple concatenation of database-only and SAN-only modules. Only low overhead monitoring data is used in the entire process.

Machine learning + domain knowledge: DIADS's workflow is a novel combination of elements from machine learning with the use of domain knowledge. A number of modules in the workflow use correlation analysis which is implemented using machine learning; the details are in Sections 4.1 and 4.2. Domain knowledge is incorporated into the workflow in Modules DA, SD, and IA; the details are given respectively in Sections 4.2–4.4. (Domain knowledge is also used in Module PD which is beyond the scope of this paper.) As we will demonstrate, the combination of machine learning and domain knowledge provides built-in checks and balances to deal with the challenges listed in Section 1.

4 Modules in the Workflow

We now provide details for all modules in DIADS's diagnosis workflow. Upfront, we would like to point out that our main goal is to describe an end-to-end instantiation of the workflow. We expect that the specific implementation techniques used for the modules will change with time as we gain more experience with DIADS.

4.1 Identifying Correlated Operators

Objective: Given a plan P that is involved in both satisfactory and unsatisfactory runs of the query, DIADS's objective in this module is to find the set of correlated operators. Let O_1, O_2, \dots, O_n be the set of all operators in P . The correlated operators form the subset of O_1, \dots, O_n whose change in running time best explains the change in P 's running time (i.e., P 's slowdown).

Technique: DIADS identifies the correlated operators by analyzing the monitoring data collected during satisfactory and unsatisfactory runs of P . This data can be seen as records with attributes $A, t(P), t(O_1), t(O_2), \dots, t(O_n)$ for each run of P .

Here, attribute $t(P)$ is the total time for one complete run of P , and attribute $t(O_i)$ is the running time of operator O_i for that run. Attribute A is an *annotation* (or *label*) associated with each record that represents whether the corresponding run of P was satisfactory or not. Thus, A takes one of two values: satisfactory (denoted S) or unsatisfactory (denoted U).

Let the values of attribute $t(O_i)$ in records with annotation S be s_1, s_2, \dots, s_k , and those with annotation U be u_1, u_2, \dots, u_l . That is, s_1, \dots, s_k are k observations of the running time of operator O_i when the plan P ran satisfactorily. Similarly, u_1, u_2, \dots, u_l are l observations of the running time of O_i when the running time of P was unsatisfactory. DIADS pinpoints correlated operators by characterizing how the distribution of s_1, \dots, s_k differs from that of u_1, \dots, u_l . For this purpose, DIADS uses *Kernel Density Estimation (KDE)* [22].

KDE is a non-parametric technique to estimate the probability density function of a random variable. Let S_i be the random variable that represents the running time of operator O_i when the overall plan performance is satisfactory. KDE applies a kernel density estimator to the k observations s_1, \dots, s_k of S_i to learn S_i 's probability density function $f_i(S_i)$.

$$f_i(S_i) = \frac{\sum_{j=1}^k K(\frac{S_i - s_j}{h})}{kh} \quad (1)$$

Here, K is a *kernel function* and h is a smoothing parameter. A typical kernel is the standard Gaussian function $K(x) = \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}}$. (Intuitively, kernel density estimators are a generalization and improvement over *histograms*.)

Let u be an observation of operator O_i 's running time when the plan performance was unsatisfactory. Consider the probability estimate $\text{prob}(S_i \leq u) = \int_{-\infty}^u f_i(S_i) ds_i$. Intuitively, as u becomes higher than the typical range of values of S_i , $\text{prob}(S_i \leq u)$ becomes closer to 1. Thus, a high value of $\text{prob}(S_i \leq u)$ represents a significant increase in the running time of operator O_i when plan performance was unsatisfactory compared to that when plan performance was satisfactory.

Specifically, DIADS includes O_i in the set of correlated operators if $\text{prob}(S_i \leq \bar{u}) \geq 1 - \alpha$. Here, \bar{u} is the average of u_1, \dots, u_l and α is a small positive constant. $\alpha = 0.1$ by default. For obvious reasons, $\text{prob}(S_i \leq \bar{u})$ is called the *anomaly score* of operator O_i .

4.2 Dependency Analysis

Objective: This module takes the set of correlated operators as input, and finds the set of system components that show a change in performance correlating with the change in running time of one of more correlated operators.

Technique: DIADS implements this module using *dependency analysis* which is based on generating and

pruning *dependency paths* for the correlated operators. We describe the generation and pruning of dependency paths in turn.

Generating dependency paths: The dependency path of an operator O_i is the set of physical (e.g., server CPU, database buffer cache, disk) and logical (e.g., volume, external workload) components in the system whose performance can have an impact on O_i 's performance. DIADS generates dependency paths automatically based on the following data:

- System-wide configuration and connectivity data as well as updates to this data collected during the execution of each operator (recall Section 3).
- Domain knowledge of how each database operator executes. For example, the dependency path of a sort operator that creates temporary tables on disk will be different from one that does not create temporaries.

We distinguish between *inner* and *outer* dependency paths. The performance of components in O_i 's inner dependency path can affect O_i 's performance directly. O_i 's outer dependency path consists of components that affect O_i 's performance indirectly by affecting the performance of components on the inner dependency path. As an example, the inner dependency path for the Index Scan operator in Figure 1 includes the server, HBA, FC-Switches, Pool2, Volume v_2 , and Disks 5-8. The outer dependency path will include Volumes v_1 and v_3 (because of the shared disks) and other database queries.

Pruning dependency paths: The fact that a component C is in the dependency path of an operator O_i does not necessarily mean that O_i 's performance has been affected by C 's performance. After generating the dependency paths conservatively, DIADS prunes these paths based on correlation analysis using KDE.

Recall from Section 3 that the monitoring data collected by DIADS contains multiple observations of the running time of operator O_i both when the overall plan ran satisfactorily and when the plan ran unsatisfactorily. For each run of O_i , consider the performance data collected by DIADS for each component C in O_i 's dependency path; this data is collected in the $[t_b, t_e]$ time interval where t_b and t_e are respectively O_i 's (absolute) start and stop times for that run. Across all runs, this data can be represented as a table with attributes $A, t(O_i), m_1, \dots, m_p$. Here, $m_1 - m_p$ are performance metrics of component C , and the annotation attribute A represents whether O_i 's running time $t(O_i)$ was satisfactory or not in the corresponding run. It follows from Section 4.1 that we can set A 's value in a record to U (denoting unsatisfactory) if $\text{prob}(S_i \leq t(O_i)) \geq 1 - \alpha$; and to S otherwise.

Given the above annotated performance data for an $\langle O_i, C \rangle$ operator-component pairing, we can apply cor-

	symp ₁	symp ₂	symp ₃	symp ₄
R ₁	1	0	0	1
R ₂	1	1	0	0
R ₃	1	0	1	1

Figure 4: Example Codebook

relation analysis using KDE to identify C 's performance metrics that are correlated with the change in O_i 's performance. The details are similar to that in Section 4.1 except for the following: for some performance metrics, observed values lower than the typical range are anomalous. This correlation can be captured using the condition $\text{prob}(M \leq v) \leq \alpha$, where M is the random variable corresponding to the metric, v is a value observed for M , and α is a small positive constant.

In effect, the dependency analysis module will identify the set of components that: (i) are part of O_i 's dependency path, and (ii) have at least one performance metric that is correlated with the running time of a correlated operator O_i . By default, DIADS will only consider the components in the inner dependency paths of correlated operators. However, components in the outer dependency paths will be considered if required by the symptoms database (Module SD).

Recall Module CR in the diagnosis workflow where DIADS checks for significant correlation between plan P 's running time and the record counts of P 's operators. DIADS implements this module using KDE in a manner almost similar to the use of KDE in dependency analysis; hence Module CR is not discussed further.

4.3 Symptoms Database

The modules so far in the workflow drilled down from the level of the query to that of physical and logical components in the system; in the process identifying correlated operators and performance metrics. While this information is useful, the detected correlations may only be *symptoms* of the true root cause(s) of the query slowdown. This issue, which can mask the true root cause(s), is generally referred to as the *event (fault) propagation* problem in diagnosis. For example, a change in data properties at the database level may, in turn, propagate to the volume level causing volume contention, and to the server level increasing CPU utilization. In addition, some spurious correlations may creep in and manifest themselves as unexpected symptoms in spite of our careful drill down process.

Objective: DIADS's Module SD tries to map the observed symptoms to the actual root cause(s), while dealing with missing as well as unexpected symptoms arising from the noisy nature of production systems.

Technique: DIADS uses a *symptoms database* to do the mapping. This database streamlines the use of domain knowledge in the diagnosis workflow to:

- Generate more accurate diagnosis results by dealing with event propagation.
- Generate diagnosis results that are semantically more meaningful to administrators (for example, reporting lock contention as the root cause instead of reporting some correlated metrics only).

We considered a number of formats proposed previously in the literature to input domain knowledge for aiding diagnosis. Our evaluation criteria were the following:

- I. How easy is the format for administrators to use? Here, usage includes customization, maintenance over time, as well as debugging. When a diagnosis tool pinpoints a particular cause, it is important that the administrators are able to understand and validate the tool's reasoning. Otherwise, administrators may never trust the tool enough to use it.
- II. Can the format deal with the noisy conditions in production systems, including multiple simultaneous problems, presence of spurious correlations, and missing symptoms.

One of the formats from the literature [16] is an *expert knowledge-base* of rules where each rule expresses patterns or relationships that describe symptoms, and can be matched against the monitoring data. Most of the focus in this work has been on exact matches, so this format scores poorly on Criterion II. Representing relationships among symptoms (e.g., event X will cause event Y) using deterministic or probabilistic networks like Bayesian networks [23] has been gaining currency recently. This format has high expressive power, but remains a black-box for administrators who find it hard to interpret the reasoning process (Criterion I).

Another format, called the *Codebook* [34], is very intuitive as well as implemented in a commercial product. This format assumes a finite set of symptoms such that each distinct root cause R has a unique *signature* in this set. That is, there is a unique subset of symptoms that R gives rise to which differs makes it distinguishable from all other root causes. This information is represented in the Codebook which is a matrix whose columns correspond to the symptoms and rows correspond to the root causes. A cell is mapped to 1 if the corresponding root cause should show the corresponding symptom; and to 0 otherwise. Figure 4 shows an example Codebook where there are four hypothetical symptoms symp_1 – symp_4 and three root causes R_1 – R_3 .

When presented with a vector V of symptoms seen in the system, the Codebook computes the *distance* $d(V, R)$ of V to each row R (i.e., root cause). Any number of different distance metrics can be used, e.g., Euclidean (L_2) distance or Hamming distance [34]. $d(V, R)$ is a measure of the confidence that R is a root cause of the problem. For example, given a symptoms vector $\langle 1, 0, 0, 1 \rangle$ (i.e., only symp_1 and symp_4 are seen), the Euclidean

distances to the three root causes in Figure 4 are 0, $\sqrt{2}$, and 1 respectively. Hence, R_1 is the best match.

The Codebook format does well on both our evaluation criteria. Codebooks can handle noisy situations, and administrators can easily validate the reasoning process. However, DIADS needs to consider complex symptoms such as symptoms with temporal properties. For example, we may need to specify a symptom where a disk failure is seen within X minutes of the first incidence of the query slowdown, where X may vary depending on the installation. Thus, it is almost impossible in our domain to fully enumerate a closed space of relevant symptoms, and to specify for each root cause whether each symptom from this space will be seen or not. These observations led to DIADS’s new design of the symptoms database:

1. We define a *base set* of symptoms consisting of: (i) operators in the database system that can be included in the correlated set, (ii) performance metrics of components that can be correlated with operator performance, and (iii) system-monitored and user-defined events collected by DIADS.
2. The language defined by IBM’s *Active Correlation Technology (ACT)* is used to express complex symptoms over the base set of symptoms [2]. The benefit of this language comes from its support for a range of built-in patterns including filter, collection, duplicate, computation, threshold, sequence, and timer. ACT can express symptoms like: (i) the workload on a volume is higher than 200 IOPS, and (ii) event E_1 should follow event E_2 in the 30 minutes preceding the first instance of query slowdown.
3. DIADS’s symptoms database is a collection of root cause entries each of which has the format $Cond_1 \& Cond_2 \& \dots \& Cond_z$, for some $z > 0$ which can differ across entries. Each $Cond_i$ is a Boolean condition of the form $\exists symp_j$ (denoting presence of $symp_j$) or $\neg \exists symp_j$ (denoting absence of $symp_j$). Here, $symp_j$ is some base or complex symptom. Each $Cond_i$ is associated with a weight w_i such the sum of the weights for each individual root cause entry is 100%. That is, $\sum_{i=1}^z w_i = 100\%$.
4. Given a vector of base symptoms, DIADS computes a *confidence score* for each root cause entry R as the sum of the weights of R ’s conditions that evaluate to true. Thus, the confidence score for R is a value in $[0\%, 100\%]$ equal to $\sum_{i=1}^z w_i |Cond_i = true$.

DIADS’s symptoms database tries to balance the expressive power of rules with the intuitive structure and robustness of Codebooks. The symptoms database differs from conventional Codebooks in a number of ways. For each

root cause entry, DIADS avoids the “closed-world” assumption for symptoms by mapping symptoms to 0, 1, or “don’t care”. Conventional Codebooks are constrained to 0 or 1 mappings. DIADS’s symptoms database can contain mappings for *fixes* to problems in addition to root causes. This feature is useful because it may be easier to specify a fix for a query slowdown (e.g., add an index) instead of trying to find the root cause. DIADS also allows multiple distinct entries for the same root cause.

Generation of the symptoms database: Companies like EMC, IBM, HP, and Oracle are investing significant (currently, mostly manual) effort to create symptoms databases for different subsystems like networking infrastructure, application servers, and databases [34, 19, 24, 9, 10, 11]. Symptoms databases created by some of these efforts are already in commercial use. The creation of these databases can be partially automated, e.g., through a combination of fault injection and machine learning [9, 12]. In fact, DIADS’s modules like correlation, dependency, and impact analysis can be used to identify important symptoms automatically.

4.4 Impact Analysis

Objective: The confidence score computed by the symptoms database module for a potential root cause R captures how well the symptoms seen in the system match the expected symptoms of R . For each root cause R whose confidence score exceeds a threshold, the impact analysis module computes R ’s *impact score*. If R is an actual root cause, then R ’s impact score represents the fraction of the query slowdown that can be attributed to R individually. DIADS’s novel impact analysis module serves three significant purposes:

- When multiple problems coexist in the system, impact analysis can separate out high-impact causes from the less significant ones; enabling prioritization of administrator effort in problem solving.
- As a safeguard against misdiagnoses caused by spurious correlations due to noise.
- As an extra check to find whether we have identified the right cause(s) or all cause(s).

Technique: Interestingly, one approach for impact analysis is to *invert* the process of dependency analysis from Section 4.2. Let R be a potential root cause whose impact score needs to be estimated:

1. Identify the set of components, denoted $comp(R)$, that R affects in the inner dependency path of the operators in the query plan. DIADS gets this information from the symptoms database.
2. For each component $C \in comp(R)$, find the subset of correlated operators, denoted $op(R)$, such that for each operator O in this subset: (i) C is in O ’s inner dependency path, and (ii) at least one performance metric of C is correlated with the change in

O 's performance. DIADS has already computed this information in the dependency analysis module.

3. R 's impact score is the percentage of the change in plan running time (query slowdown) that can be attributed to the change in running time of operators in $op(R)$. Here, change in running time is computed as the difference between the average running times when performance is unsatisfactory and that when performance is satisfactory.

The above approach will work as long as for any pair of suspected root causes R_1 and R_2 , $op(R_1) \cap op(R_2) = \emptyset$. However, if there are one or more operators common to $op(R_1)$ and $op(R_2)$ whose running times have changed significantly, then the above approach cannot fully separate out the individual impacts of R_1 and R_2 .

DIADS addresses the above problem by leveraging *plan cost models* that play a critical role in all database systems. For each query submitted to a database system, the system will consider a number of different plans, use the plan cost model to predict the running time (or some other cost metric) of each plan, and then select the plan with minimum predicted running time to run the query to completion. These cost models have two main components:

- Analytical formula per operator type (e.g., sort, index scan) that estimates the resource usage (e.g., CPU and I/O) of the operator based on the values of input parameters. While the number and types of input parameters depend on the operator type, the main ones are the sizes of the input processed by the operator.
- Mapping parameters that convert resource-usage estimates into running-time estimates. For example, IBM DB2 uses two such parameters to convert the number of estimated I/Os into a running-time estimate: (i) the overhead per I/O operation, and (ii) the transfer rate of the underlying storage device.

The following are two examples of how DIADS uses plan cost models:

- Since DIADS collects the old and new record-counts for each operator, it estimates the impact score of a change in data properties by plugging the new record-counts into the plan cost model.
- When volume contention is caused by an external workload, DIADS estimates the new I/O latency of the volume from actual observations or the use of device performance models. The impact score of the volume contention is computed by plugging this new estimate into the plan cost model.

DIADS's use of plan cost models is a general technique for impact analysis, but it is limited by what effects are accounted for in the model. For example, if wait times for locks are not modeled, then the impact score cannot be computed for locking-based problems. Addressing this issue—e.g., by extending plan cost models or by

using *planned experiments* at run time—is an interesting avenue for future work.

5 Experimental Evaluation

The taxonomy of scenarios considered for diagnosis in the evaluation follows from Figure 2. DIADS was used to diagnose query slowdowns caused by (i) events within the database and the SAN layers, (ii) combinations of events across both layers, as well as (iii) multiple concurrent problems (a capability unique to DIADS). Due to space limitations, it is not possible to describe all the scenario permutations from Figure 2. Instead, we start with a scenario and make it increasingly complex by combining events across the database and SAN. We consider: (i) volume contention caused by SAN misconfiguration, (ii) database-level problems (change in data properties, contention due to table locking) whose symptoms propagate to the SAN, and (iii) independent and concurrent database-level and SAN-level problems.

We provide insights into how DIADS diagnoses these problems by drilling down to the intermediate results like anomaly, confidence, and impact scores. While there is no equivalent tool available for comparison with DIADS, we provide insights on the results that a database-only or SAN-only tool would have generated; these insights are derived from hands-on experience with multiple in-house and commercial tools used by administrators today. Within the context of the scenarios, we also report sensitivity analysis of the anomaly score to the number of historic samples and length of the monitoring interval.

5.1 Setup Details

Our experimental testbed is part of a production SAN environment, with the interconnecting fabric and storage controllers being shared by other applications. Our experiments ran during low activity time-periods on the production environment. The testbed runs data-warehousing queries from the popular TPC-H benchmark [29] on a PostgreSQL database server configured to access tables using two Ext3 filesystem volumes created on an enterprise-class IBM DS6000 storage controller. The database server is a 2-way 1.7 GHz IBM xSeries machine running Linux (Redhat 4.0 Server), connected to the storage controller via Fibre Channel (FC) host bus adaptor (HBA). Both the storage volumes are RAID 5 configurations consisting of (4 + 2P) 15K FC disks.

An IBM TotalStorage Productivity Center [17] SAN management server runs on a separate machine recording configuration details, statistics, and events from the SAN as well as from PostgreSQL (which was instrumented to report the data to the management tool). Figure 6 shows the key performance metrics collected from the database and SAN. The monitoring data is stored as time-series data in a DB2 database. Each module in DI-

ADS's workflow is implemented using a combination of Matlab scripts (for KDE) and Java. DIADS uses a symptoms database that was developed in-house to diagnose query slowdowns in database over SAN deployments.

Our experimental results focus on the slowdown of the plan shown in Figure 5 for Query 2 from TPC-H. Figure 5 shows the 25 operators in the plan, denoted O_1 – O_{25} . In database terminology, the operators Index Scan and Sequential Scan are *leaf* operators since they access data directly from the tables; hence the leaf operators are the most sensitive to changes in SAN performance. The plan has 9 leaf operators. The other operators process intermediate results.

5.2 Scenario 1: Volume Contention due to SAN Misconfiguration

Problem Setting

In this scenario, a contention is created in volume V1 (from Figure 5) causing a slowdown in query performance. The root cause of the contention is another application workload that is configured in the SAN to use a volume V' that gets mapped to the same physical disks as V1. For an accurate diagnosis result, DIADS needs to pinpoint the combination of SAN configuration events generated on: (i) creation of the new volume V', and (ii) creation of a new zoning and mapping relationship of the server running the workload that accesses V'.

Module CO

DIADS analyzes the historic monitoring samples collected for each of the 25 query operators. The monitoring samples for an operator are labeled as satisfactory or unsatisfactory based on past problem reports from the administrator. Using the operator running times in these labeled samples, Module CO in the workflow uses KDE to compute anomaly scores for the operators (recall Section 4.1). Table 1 shows the anomaly scores of the operators identified as the correlated operators; these operators have anomaly scores ≥ 0.8 (the significance of the anomaly scores is covered in Section 4.1). The following observations can be made from Table 1:

- Leaf operators O_8 and O_{22} were correctly identified as correlated. These two are the only leaf operators that access data on the Volume V1 under contention.
- Eight intermediate operators were ranked highly as well. This ranking can be explained by event propagation where the running times of these operators are affected by the running times of the “upstream” operators in the plan (in this case O_8 and O_{22}).
- A false positive for leaf operator O_4 which operates on tables in Volume V2. This could be a result of noisy monitoring data associated with the operator.

In summary, Module CO's KDE analysis has zero false negatives and one false positive from the total set of 9

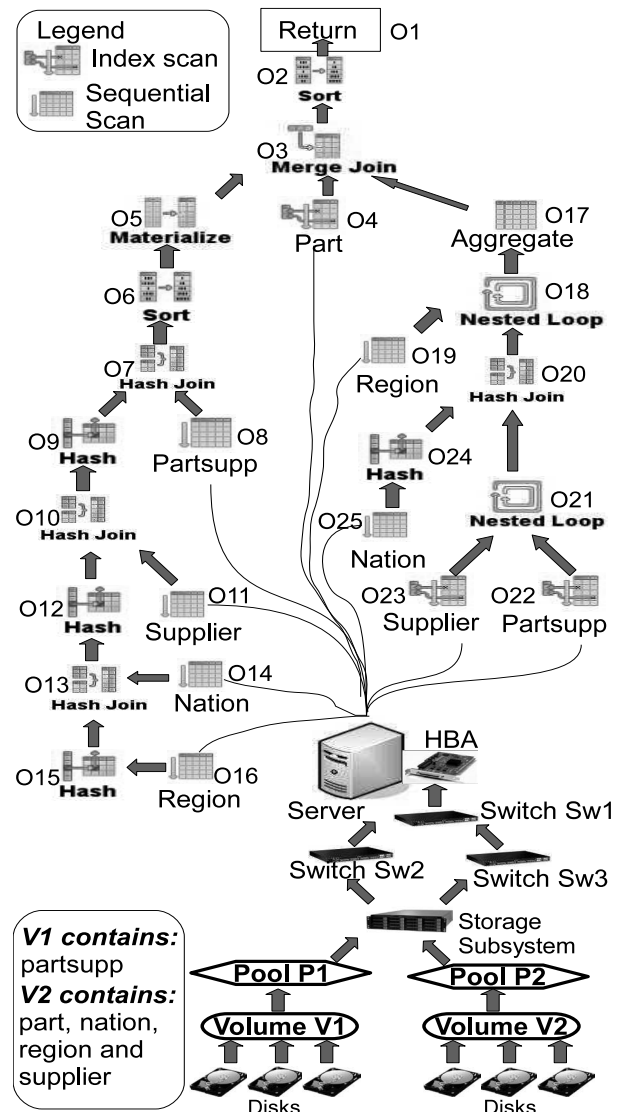


Figure 5: Query plan, operators, and dependency paths for the experimental results

leaf operators. The false positive gets filtered out later in the symptoms database and impact analysis modules.

To further understand the anomaly scores, we conducted a series of sensitivity tests. Figure 7 shows the sensitivity of the anomaly scores of three representative operators to the number of samples available from the satisfactory runs. O_{22} 's score converges quickly to 1 because O_{22} 's running time under volume contention is almost 5X the normal. However, the scores for leaf operator O_{11} and intermediate operator O_1 take around 20 samples to converge. With fewer than these many samples, O_{11} could have become a false positive. In all our results, the anomaly scores of all 25 operators converge within 20 samples. While more samples may be required in environments with higher noise levels, the relative simplicity of KDE (compared to models like Bayesian

Database Metrics	Server Metrics	Network Metrics	Storage Metrics
Operator Start Stop Times Record-counts Plan Start Stop Times Locks outstanding and held Lock wait times Space Usage Blocks Read Buffer Hits Index Scans Index Reads Index Fetches Sequential Scans	CPU Usage (%ge) CPU Usage (Mhz) Handles Threads Processes Heap Memory Usage(KB) Physical Memory Usage (%) Kernel Memory(KB) Memory Being Swapped(KB) Reserved Memory Capacity(KB) Wait I/O Network Bandwidth (HBA)	Bytes Transmitted Bytes Received Packets Transmitted Packets Received LIP Count NOS Count Error Frames Dumped Frames Link Failures CRC Errors Address Errors	Bytes Read Bytes Written Contaminating Writes PhysicalStorageRead Operations Physical Storage Read Time PhysicalStorageWriteOperations Physical Storage Write Time Sequential Read Requests Sequential Write Requests Total IOs

Figure 6: Important performance metrics collected by DIADS

Operator	Operator Type	Anomaly Score
O_2	Non-leaf	1.00
O_3	Non-leaf	1.00
O_6	Non-leaf	1.00
O_7	Non-leaf	1.00
O_8	Leaf (sequential scan)	1.00
O_{18}	Non-leaf	1.00
O_{20}	Non-leaf	1.00
O_{21}	Non-leaf	1.00
O_{22}	Leaf (index scan)	1.00
O_{17}	Non-leaf	0.969
O_4	Leaf (index scan)	0.965

Table 1: Anomaly scores for query operators from Figure 5 in Scenario 1

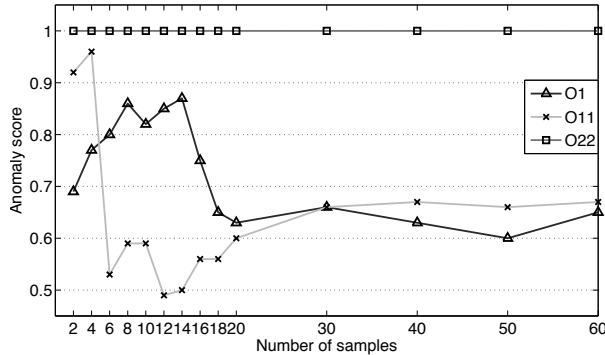


Figure 7: Sensitivity of anomaly scores to the number of satisfactory samples. While O_{22} shows highly anomalous behavior, scores for O_1 and O_{11} should be low

networks) keeps this number low.

Figure 8 shows the sensitivity of O_{22} 's anomaly score to the length of the monitoring interval during a 4-hour period. Intuitively, larger monitoring intervals suppress the effect of spikes and bursty access patterns. In our experiments, the query running time was around 4 minutes under satisfactory conditions. Thus, monitoring intervals of 10 minutes and larger in Figure 8 cause the anomaly score to deviate more and more from the true value.

Module DA

This module generates and prunes dependency paths for correlated operators in order to relate operator perfor-

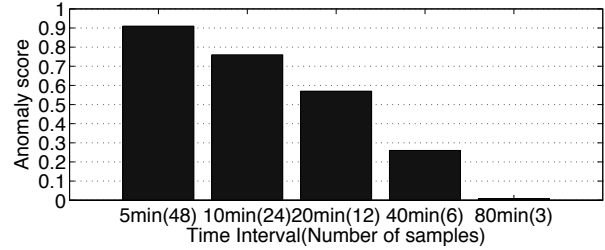


Figure 8: Sensitivity of anomaly scores to noise in the monitoring data

Volume, Perf. Metric	Anomaly Score (no contention in V2)	Anomaly Score (contention in V2)
V1, writeIO	0.894	0.894
V1, writeTime	0.823	0.823
V2, writeIO	0.063	0.512
V2, writeTime	0.479	0.879

Table 2: Anomaly scores computed during dependency analysis for performance metrics from Volumes V1, V2

mance to database and SAN component performance. For ease of presentation, we will focus on the leaf operators in Figure 5 since they are the most sensitive to SAN performance. Given the configuration of our experimental testbed in Figure 5, the primary difference between the dependency paths of various operators is in the volumes they access: V1 is in the dependency path of O_8 and O_{22} , and V2 is in the paths of O_4 , O_{11} , O_{14} , O_{16} , O_{19} , O_{23} , and O_{25} .

The set of correlated operators from Module CO are O_4 , O_8 , and O_{22} . Thus, DIADS will compute anomaly scores for the performance metrics of both V1 and V2. Table 2's second column shows the anomaly scores for two representative metrics each from V1 and V2. (Table 2's third column is described later in this section.) As expected, none of V2's metrics are identified as correlated because V2 has no contention; while those of V1 are.

Module CR

Anomaly scores are low in this module because data properties do not change.

Module SD

The symptoms identified up to this stage are:

- High anomaly scores for operators dependent on V1.
- High anomaly scores for V1's performance metrics.
- High anomaly score for only one V2-dependent operator (out of seven such operators).

These symptoms are strong evidence that V1's performance is a cause of the query slowdown, and V2's performance is not. Thus, even when a symptoms database is not available, DIADS correctly narrows down the search space an administrator has to consider during diagnosis. An impact analysis will further point out that the false positive symptom due to O_4 has little impact on the query slowdown.

However, without a symptoms database or further diagnosis effort from the administrator, the *root cause* of V1's change of performance is still unknown among possible candidates like: (i) change of performance of an external workload, (ii) a runaway query in the database, or (iii) a RAID rebuild. We will now report results from the use of a symptoms database that was developed in-house. DIADS uses this database as described in Section 4.3 except that instead of reporting numeric confidence scores to administrators, DIADS reports confidence as one of High ($score \geq 80\%$), Medium ($80\% > score \geq 50\%$), or Low ($50\% > score \geq 0\%$). The summary of Module SD's output in the current scenario is:

- All root causes with contention-related symptoms for V2 have Low confidence (few symptoms are found).
- RAID rebuild gets Low confidence because no RAID rebuild start or end events are found.
- V1 contention due to changes in data properties gets Low confidence because symptoms are missing.
- V1 contention due to change in external workload gets Low confidence because no external workload was on the outer dependency path of a correlated operator when performance was satisfactory.
- V1 contention due to change in database workload gets Medium confidence because of a weak correlation between the performance of some correlated operators and the rest of the database workload.
- V1 contention due to the SAN misconfiguration problem gets High confidence because all specified symptoms are found including: (i) creation of a new volume (parametrized with the physical disk information), and (ii) creation of new masking and zoning information for the volume.

The symptoms database had an entry for the actual root cause because this problem is common. Hence, DIADS was able to diagnose the root cause for this scenario. Note that DIADS had to consider more than 900 events (system generated as well as user-defined) for the database and SAN generated during the course of the sat-

isfactory and unsatisfactory runs for this experiment.

Module IA

Impact analysis done using the inverse dependency analysis technique gives an impact score of 99.8% for the high-confidence root cause found. This score is high because the slowdown is caused entirely by the contention in V1.

In keeping with our experimental methodology, we complicated the problem scenario to test DIADS's robustness. Everything was kept the same except that we created extra I/O load on Volume V2 in a bursty manner such that this extra load had little impact on the query beyond the original impact of V1's contention. Without intrusive tracing, it would not be possible to rule out the extra load on V2 as a potential cause of the slowdown.

Interestingly, DIADS's integrated approach is still able to give the right answer. Compared to the previous scenario, there will now be some extra symptoms due to higher anomaly scores for V2's performance metrics (as shown in the third column in Table 2). However, root causes with contention-related symptoms for V2 will still have Low confidence because most of the leaf operators depending on V2 will have low anomaly scores as before. Also, impact scores will be low for these causes.

Unlike DIADS, a SAN-only diagnosis tool may spot higher I/O loads in both V1 and V2, and attribute both of these as potential root causes. Even worse, the tool may give more importance to V2 because most of the data is on V2. A database-only tool can pinpoint the slowdown in the operators. However, this tool cannot track the root cause down to the SAN level because it has no visibility into SAN configuration or performance. From our experience, database-only tools may give several false positives in this context, e.g., suboptimal bufferpool setting or a suboptimal choice of execution plan.

5.3 Scenario 2: Database-layer Problem Propagating to the SAN-layer

In this scenario we cause a query slowdown by changing the properties of the data, causing extra I/O on Volume V2. The change is done by an update statement that modifies the value of an attribute in some records of the *part* table. The overall size of all tables, including *part*, are unchanged. There are no external causes of contention on the volumes.

Modules CO, DA, and CR behave as expected. In particular, module CR correctly identifies all the operators whose record-counts show a correlation with plan performance: operators O_1 , O_2 , O_3 , and O_4 show increased record-counts, while operators O_5 and O_6 show reduced record-counts. The root-cause entry for changes in data properties gets High confidence in Module SD because all needed symptoms match. All other root-cause entries

get Low confidence, including contention due to changes in external workload and database workload because no correlations are detected on the outer dependency paths of correlated operators (as expected).

The impact analysis module gives the final confirmation that the change in data properties is the root cause, and rules out the presence of high-impact external causes of volume contention. As described in Section 4.4, we can use the plan cost model from the database to estimate the individual impact of any change in data properties. In this case, the impact score for the change in data properties is 88.31%. Hence, DIADS could have diagnosed the root cause of this problem even if the symptoms database was unavailable or incomplete.

5.4 Scenario 3: Concurrent Database-layer and SAN-layer Problems

We complicate Scenario 2 by injecting contention on Volume V2 due to SAN misconfiguration along with the change in data properties. Both these problems individually cause contention in V2. The SAN misconfiguration is the higher-impact cause in our testbed. This key scenario represents the occurrence of multiple, possibly related, events at the database and SAN layers, complicating the diagnosis process. The expected result from DIADS is the ability to pinpoint both these events as causes, and giving the relative impact of each cause on query performance.

The CO, DA, and CR Modules behave in a fashion similar to Scenario 2, and drill down to the contention in Volume V2. We considered DIADS's performance in two cases: with and without the symptoms database. When the symptoms database is unavailable or incomplete, DIADS cannot distinguish between Scenarios 2 and 3. However, DIADS's impact analysis module computes the impact score for the change in data properties, which comes to 0.56%. (This low score is representative because the SAN misconfiguration has more than 10X higher impact on the query performance than the change in data properties.) Hence, DIADS final answer in this case is as follows: (i) a change in data properties is a high-confidence but low-impact cause of the problem, and (ii) there are one or more other causes that impact V2 which could not be diagnosed.

When the symptoms database is present, both the actual root causes are given High confidence by Module SD because the needed symptoms are seen in both cases. Thus, DIADS will pinpoint both the causes. Furthermore, impact analysis will confirm that the full impact on the query performance can be explained by these two causes.

A database-only diagnosis tool would have successfully diagnosed the change in data properties in both Scenarios 2 and 3. However, the tool may have difficulty distinguishing between these two scenarios or pinpoint-

ing causes at the SAN layer. A SAN-only diagnosis tool will pinpoint the volume overload. However, it will not be able to separate out the impacts of the two causes. Since the sizes of the tables do not change, we also suspect that such a tool may even rule out the possibility of a change in data properties being a cause.

5.5 Discussion

The scenarios described in the experimental evaluation were carefully chosen to be simple, but not simplistic. They are representative of event categories occurring within the DB and SAN layers as shown in Figure 2. We have additionally experimented with different events within those categories such as CPU and memory contention in the SAN in addition to disk-level saturation, different types of database misconfiguration, and locking-based database problems. Locking-based problems are hard to diagnose because they can cause different types of symptoms in the SAN layer, including contention as well as underutilization. We have also considered concurrent occurrence of three or more problems, e.g., change in data properties, SAN misconfiguration, and locking-based problems. The insights from these experiments are similar to those seen already, and further confirm the utility of an integrated tool. However:

- High levels of noise in the monitoring data can reduce DIADS's effectiveness.
- While DIADS would still be effective when the symptoms database is incomplete, more manual effort will be needed to pinpoint actual root causes.
- Incomplete or inaccurate plan cost models reduce the accuracy of impact analysis.

6 Conclusions and Future Work

We presented an integrated database and storage diagnosis tool called DIADS. Using a novel combination of machine learning techniques with database and storage expert domain-knowledge, DIADS accurately identifies the root cause(s) of problems in query performance; irrespective of whether the problem occurs in the database or the storage layer. This integration enables a more accurate and efficient diagnosis tool for system administrators. Through a detailed experimental evaluation, we also demonstrated the robustness of our approach: with its ability to deal with concurrent multiple problems as well as presence of noisy data.

In future, we are interested in exploring two directions of research. First, we are investigating approaches that further strengthen the analysis done as part of DIADS modules, e.g., techniques that complement database query plan models using planned run-time experiments. Second, we aim to generalize our diagnosis techniques to support applications other than databases in conjunction with enterprise storage.

References

- [1] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 74–89.
- [2] ANA BIAZZETTI AND KIM GAJDA. *Achieving Complex Event Processing with Active Correlation Technology*. <http://www.ibm.com/developerworks/library/ac-acact/index.html>.
- [3] ARANYA, A., WRIGHT, C. P., AND ZADOK, E. Tracefs: A File System to Trace Them All. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)* (2004), pp. 129–145.
- [4] BASU, S., DUNAGAN, J., AND SMITH, G. Why Did My PC Suddenly Slow Down? In *Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques (SYSML)* (2007), pp. 1–6.
- [5] BORISOV, N., UTTAMCHANDANI, S., ROUTRAY, R., AND SINGH, A. Why Did My Query Slow Down? In *Proceedings of the Fourth Biennial Conference on Innovative Data Systems Research (CIDR)* (2009).
- [6] CHAUDHURI, S., KÖNIG, A. C., AND NARASAYYA, V. R. SQLCM: A Continuous Monitoring Framework for Relational Database Engines. In *Proceedings of the International Conference on Data Engineering (ICDE)* (2004), pp. 473–485.
- [7] CHEN, M. Y., ACCARDI, A., KICIMAN, E., LLOYD, J., PATTERSON, D., FOX, A., AND BREWER, E. Path-based Failure and Evolution Management. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)* (2004), pp. 23–23.
- [8] COHEN, I., CHASE, J. S., GOLDSZMIDT, M., KELLY, T., AND SYMONS, J. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proceedings of the USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (Dec. 2004), pp. 104–109.
- [9] COHEN, I., ZHANG, S., GOLDSZMIDT, M., SYMONS, J., KELLY, T., AND FOX, A. Capturing, Indexing, Clustering, and Retrieving System History. In *ACM symposium on Operating systems principles (SOSP)* (2005), pp. 105–118.
- [10] DAGEVILLE, B., DAS, D., DIAS, K., YAGHOUB, K., ZAIT, M., AND ZIAUDDIN, M. Automatic SQL Tuning in Oracle 10g. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)* (2004), pp. 1098–1109.
- [11] DIAS, K., RAMACHER, M., SHAFT, U., VENKATARAMANI, V., AND WOOD, G. Automatic Performance Diagnosis and Tuning in Oracle. In *Proceedings of Conference on Innovative Data Systems Research (CIDR)* (2005), pp. 84–94.
- [12] DUAN, S., BABU, S., AND MUNAGALA, K. Fa: A System for Automating Failure Diagnosis. In *Proceedings of the International Conference on Data Engineering (ICDE)* (Apr. 2009).
- [13] EMC CONTROL CENTER FAMILY. <http://www.emc.com/products/storage.management/controlcenter.jsp>.
- [14] GARCIA-MOLINA, H., ULLMAN, J., AND WIDOM, J. *Database Systems: The Complete Book*. Prentice Hall, Upper Saddle River, New Jersey, 2001.
- [15] HEWLETT PACKARD SYSTEMS INSIGHT MANAGER. <http://h18002.www1.hp.com/products/servers/management/hpsim/index.html>.
- [16] IBM TIVOLI NETWORK MANAGER. <http://www-01.ibm.com/software/tivoli/products/netcool-precision-ip>.
- [17] IBM TOTALSTORAGE PRODUCTIVITY CENTER. <http://www-306.ibm.com/software/tivoli/products/totalstorage-data/>.
- [18] JOUKOV, N., TRAEGER, A., IYER, R., WRIGHT, C. P., AND ZADOK, E. Operating System Profiling via Latency Analysis. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI)* (2006), pp. 89–102.
- [19] MANJI, A. Creating Symptom Databases to Rervice J2EE Applications in WebSphere Studio, 2004.
- [20] MEHTA, A., GUPTA, C., WANG, S., AND DAYAL, U. Automatic Workload Management for Enterprise Data Warehouses. *IEEE Data Eng. Bull.* 31, 1 (2008), 11–19.
- [21] MESNIER, M. P., WACHS, M., SAMBASIVAN, R. R., ZHENG, A. X., AND GANGER, G. R. Modeling the Relative Fitness of Storage. *SIGMETRICS Perform. Eval. Rev.* 35, 1 (2007), 37–48.
- [22] PARZEN, E. On Estimation of a Probability Density Function and Mode. *Ann. Math. Stat.* 33 (1962), 1065–1076.
- [23] PEARL, J. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, Cambridge, UK, 2000.
- [24] PERAZOLO, M. The Autonomic Computing Symptoms Format, 2005. IBM Library.
- [25] POLLACK, K. T., AND UTTAMCHANDANI, S. Genesis: A Scalable Self-Evolving Performance Management Framework for Storage Systems. In *IEEE International Conference on Distributed Computing Systems (ICDCS)* (2006), p. 33.
- [26] QIN, Y., SALEM, K., AND GOEL, A. K. Towards Adaptive Costing of Database Access Methods. In *Proceedings of the International Workshop on Self-Managing Database Systems (SMDb)* (2007), pp. 469–477.
- [27] REISS, F., AND KANUNGO, T. A Characterization of the Sensitivity of Query Optimization to Storage Access Cost Parameters. In *SIGMOD Conference* (2003), pp. 385–396.
- [28] SHEN, K., ZHONG, M., AND LI, C. I/O System Performance Debugging Using Model-driven Anomaly Characterization. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies (FAST)* (2005), pp. 23–23.
- [29] THE TPC-H DECISION SUPPORT BENCHMARK. www.tpc.org/tpch.
- [30] THERESKA, E., SALMON, B., STRUNK, J., WACHS, M., ABDEL-MALEK, M., LOPEZ, J., AND GANGER, G. R. Stardust: Tracking Activity in a Distributed Storage System. *SIGMETRICS Perform. Eval. Rev.* 34, 1 (2006), 3–14.
- [31] VMWARE VIRTUAL CENTER. <http://www.vmware.com/products/vi/vc/>.
- [32] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of USENIX OSDI Conference* (2004), pp. 245–258.
- [33] WEIKUM, G., MOENKEBERG, A., HASSE, C., AND ZABBACK, P. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In *Proceedings of the VLDB Conference* (Hongkong, China, 2002), pp. 20–31.
- [34] YEMINI, S. A., KLIGER, S., MOZES, E., YEMINI, Y., AND OHSIE, D. High Speed and Robust Event Correlation. *IEEE Communications Magazine* 34 (1996).
- [35] ZHANG, S., COHEN, I., SYMONS, J., AND FOX, A. Ensembles of Models for Automated Diagnosis of System Performance Problems. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)* (2005), pp. 644–653.
- [36] ZHOU, S., COSTA, H. D., AND SMITH, A. J. A File System Tracing Package for Berkeley UNIX. Tech. rep., 1985.

Dynamic Resource Allocation for Database Servers Running on Virtual Storage

Gokul Soundararajan, Daniel Lupei, Saeed Ghanbari,
Adrian Daniel Popescu, Jin Chen*, Cristiana Amza
Department of Electrical and Computer Engineering
*Department of Computer Science**
University of Toronto

Abstract

We introduce a novel multi-resource allocator to dynamically allocate resources for database servers running on virtual storage. Multi-resource allocation involves proportioning the database and storage server caches, and the storage bandwidth between applications according to overall performance goals. The problem is challenging due to the interplay between different resources, e.g., changing any cache quota affects the access pattern at the cache/disk levels below it in the storage hierarchy. We use a combination of on-line modeling and sampling to arrive at near-optimal configurations within minutes. The key idea is to incorporate access tracking and known resource dependencies e.g., due to cache replacement policies, into our performance model.

In our experimental evaluation, we use both micro-benchmarks and the industry standard benchmarks TPC-W and TPC-C. We show that our multi-resource allocation approach improves application performance by up to factors of 2.9 and 2.4 compared to state-of-the-art single-resource controllers, and their ad-hoc combination, respectively.

1 Introduction

With the emerging trend towards server consolidation in large data centers, techniques for dynamic resource allocation for performance isolation between applications become increasingly important. With server consolidation, operators multiplex several concurrent applications on each physical server of a server farm, connected to a shared network attached storage (as in Figure 1). As compared to traditional environments, where applications run in isolation on over-provisioned resources, the benefits of server consolidation are reduced costs of management, power and cooling. However, multiplexed applications are in competition for system resources, such as, CPU, memory and disk, especially during load bursts.

Moreover, in this shared environment, the system is still required to meet per-application performance goals. This gives rise to a complex resource allocation and control problem.

Currently, resource allocation to applications in state-of-the-art platforms occurs through different performance optimization loops, run independently at different levels of the software stack, such as, at the database server, operating system and storage server, in the consolidated storage environment shown in Figure 1. Each local controller typically optimizes its own local goals, e.g., hit-ratio, disk throughput, etc., oblivious to application-level goals. This might lead to situations where local, per-controller, resource allocation optima do not lead to the global optimum; indeed local goals may conflict with each other, or with the per-application goals [14]. Therefore, the main challenge in these modern enterprise environments is designing a strategy which adopts a *holistic* view of system resources; this strategy should efficiently allocate all resources to applications, and enforce per-application quotas in order to meet overall optimization goals e.g., overall application performance or service provider revenue.

Unfortunately, the general problem of finding the globally optimum partitioning of all system resources, at all levels to a given set of applications is an NP-hard problem. Complicating the problem are interdependencies between the various resources. For example, let's assume the two tier system composed of database servers and consolidated storage server as in Figure 1, and several applications running on each database server instance. For any given application, a particular cache quota setting in the buffer pool of the database system influences the number and type of accesses seen at the storage cache for that application. Partitioning the storage cache, in its turn, influences the access pattern seen at the disk. Hence, even deriving an off-line solution, assuming a stable set of applications, and available hardware e.g., through profiling, trial and

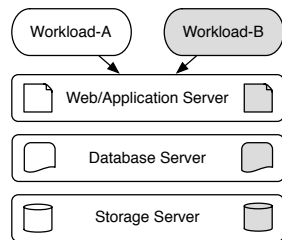


Figure 1: **Data Center Infrastructure:** We show a typical data-center architecture using consolidated storage

error, etc., by the system administrator, is likely to be highly inaccurate, time consuming, or both.

Due to these problems, with a few exceptions [17, 32], previous work has eschewed dynamic resource partitioning policies, in favor of investigating mechanisms for enforcing performance isolation, under the assumption that per-application quotas, deadlines or priorities are predefined e.g., manually, for each given resource type. Examples of such mechanisms include CPU quota enforcement [2, 16], memory quota allocation based on priorities [3], or I/O quota enforcement between workloads [9, 11, 12].

Moreover, typically, previous work investigated enforcing a given resource partitioning of a single resource, within a single software tier at a time. In our own previous work in the area of dynamic partitioning, we have investigated either partitioning memory, through a simulation-based exhaustive search approach [24], or partitioning storage bandwidth, through an adaptive feedback-loop approach [23], but not both.

In this paper, we consider the problem of global resource allocation, which involves proportioning the database and storage server caches, and the storage bandwidth among applications, according to overall performance goals. To achieve this, we focus on building a simple performance model in order to guide the search, by providing a good approximation of the overall solution. The performance model provides a resource-to-performance mapping for each application, in all possible resource quota configurations. Our key ideas are to incorporate readily available information about the application and system into the performance model, and then refine the model through limited experimental sampling of actual behavior. Specifically, we reuse and extend on-line models for workload characterization, i.e., the *miss ratio curve* (MRC) [32], as well as simplifications based on common assumptions about cache replacement policies. We further derive a disk latency model for a quantized disk scheduler [27] and we parametrize the model with metrics collected from the on-line system, instead

of using theoretical value distributions, thus avoiding the fundamental source of inaccuracy in classic analytical models [10].

Finally, we refine the accuracy of the computed performance model through experimental sampling. We use statistical interpolation between computed and experimental sample points in order to re-approximate the per-application performance models, thus dynamically refining the model. We experimentally show that, by using this method, convergence towards near-optimal configurations can be achieved in mere minutes, while an exhaustive exploration of the multi-dimensional search space, representing all possible partitioning configurations, would take weeks, or even months.

We implement our technique using commodity software and hardware components without any modifications to interfaces between components, and with minimal instrumentation. We use the MySQL database engine running a set of standard benchmarks, i.e., the TPC-W e-commerce benchmark, and the TPC-C transaction processing benchmark. Our experimental testbed is a cluster of dual processor servers connected to a commodity storage hardware.

We show experiments for on-line convergence to a global partitioning solution for sharing the database buffer pool, storage cache, and disk bandwidth in different application configurations. We compare our approach to two baseline approaches, which optimize either the memory partitioning, or the disk partitioning, as well as combinations of these approaches without global coordination. We show that for most application configurations, our computed model effectively prunes most of the search space, even without any additional tuning through experimental sampling. Our dynamic resource algorithm performs similar to an experimental exhaustive search algorithm, but provides a solution within minutes, versus days of running time. At the same time, our global resource partitioning solution improves application performance by up to factors of 2.9 and 2.4 compared to state-of-the-art single-resource controllers and their ad-hoc combination, respectively.

The remainder of this paper is structured as follows. Section 2 provides a background on existing techniques for server consolidation in modern data centers, highlighting the need for a global resource allocation solution. We describe our multi-resource partitioning algorithm in Section 3. Section 4 describes our virtual storage prototype and sampling methodology in detail. Section 5 presents the algorithms we use for comparison, our benchmarks, and our experimental methodology, while Section 6 presents the results of our experiments on this platform. Section 7 discusses related work and Section 8 concludes the paper.

2 Background and Motivation

In this section, we present and evaluate the state-of-the-art in single resource partitioning and we show why these techniques are insufficient in themselves.

2.1 Single Resource Partitioning

We describe previous work that either allocate the storage bandwidth, or cache/memory to several applications.

Storage Bandwidth Partitioning: Several disk scheduling policies [11, 12, 27, 29] for enforcing disk bandwidth isolation between co-scheduled applications have been proposed. We have implemented and compared the performance isolation guarantees provided by the following disk schedulers: (1) Quanta-based scheduling [27], (2) Start-time Fair Queuing (SFQ) [11], (3) Earliest Deadline First (EDF), (4) Lottery-based [29] and (5) Façade [12]. Our study [18] shows that the Quanta-based scheduler, where each workload is given a quantum of time for using the disk in exclusive mode, offers the best performance isolation level. This is because it allows the storage server to exploit the locality in I/O requests issued by an application during its assigned quantum, which in turn results in minimizing the effects of additional disk seeks due to inter-application interference. However, the existing algorithms discussed above assume that the I/O deadlines, or disk bandwidth proportions are given *a priori*. In this paper, we study how to dynamically determine the bandwidth proportions at run-time. Once the bandwidth proportions are determined, we use Quanta-based scheduling to enforce the allocations, since it provides the strongest isolation guarantees.

Memory/Cache Partitioning: Dynamic memory partitioning between applications is typically performed using the *miss ratio curve* (MRC) [32]. The MRC represents the page miss ratio versus the memory size, and can be computed dynamically through Mattson's Stack Algorithm [13]. The algorithm assigns memory increments iteratively to the application with the highest predicted miss ratio benefit. MRC-based cache partitioning thus dynamically partitions the cache/memory to multiple applications, in such a way to optimize the aggregate miss ratio.

2.2 Motivating Experiment

We present a simple motivating experiment that shows the need for multi-resource allocation. To simplify the presentation, we consider only accesses to the storage server, hence only the *storage cache* and the *storage bandwidth* resources. We run two synthetic workloads concurrently on the storage server: a small workload (Workload-A) with 1 outstanding request, and a large

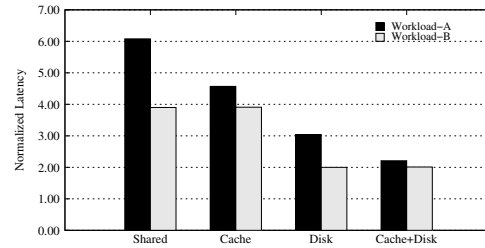


Figure 2: **Motivating Results:** Comparison of aggregate latency motivates multi-resource controllers.

workload (Workload-B) with 10 outstanding requests, at any given time. Workload-A is cache friendly and achieves a cache hit ratio of 50% with a 1GB storage cache. In contrast, Workload-B is mostly un-cacheable; it obtains only a 5% hit ratio with a 1GB storage cache.

We run the workloads using several different configurations, i.e., uncontrolled sharing, partitioning the cache, disk or both between workloads. We normalize the latency of each workload relative to its latency running in isolation. Figure 2 presents our results. In all schemes, we use the combined application latencies (by simple summation) as the global optimization goal. We choose this simple metric for fairness of comparison with the *miss ratio curve* algorithm [32], which optimizes the aggregate miss ratio, hence the aggregate latency, while being agnostic to Service Level Objectives (SLOs) in general.

When running in isolation, Workload-A is able to utilize the 1 GB cache effectively and this results in an average storage access latency of 4.4ms. On the other hand, Workload-B does not benefit from the cache, resulting in an average storage access latency of 85.1ms. When the two workloads are run concurrently with uncontrolled resource sharing, the larger Workload-B dominates the smaller Workload-A at both cache and disk levels. This results in a factor of 6 slowdown for Workload-A and a factor of 4 slowdown for Workload-B. This result shows that workloads can suffer significant performance degradation when resource sharing is not controlled.

Next, we run the workloads using different resource partitioning algorithms. First, we partition the storage cache using the *miss ratio curves* of the workloads [32], while disk bandwidth sharing is uncontrolled. The MRC algorithm determines that the best cache setting is to allocate the bulk of the storage cache (992 MB) to Workload-A and provide a minimum to Workload-B. Cache partitioning thus improves the performance of Workload-A significantly from 26.6ms to 19.9ms. Next, we iterate through all possible disk partitioning settings to find the best disk bandwidth partitioning between the workloads, and enforce it using quanta-based scheduling [27], while

cache sharing is uncontrolled. By partitioning the disk bandwidth, the performance of Workload-A improves to 13.2ms. In addition, Workload-B improves to 169.7ms. While properly partitioning the resource at each level independently, as described above, alleviates the interference, neither partitioning results in the optimal configuration for these two workloads.

On the other hand, an exhaustive search of both the cache and bandwidth settings yields an ideal setting where the storage access latency is 9.64ms for Workload-A and 171.3ms for Workload-B. In our simple case, the allocation solution found by the exhaustive search algorithm is just a combination of the solutions found by the two independent partitioners, for cache and disk. However, as we will show, due to the interdependence between resources, this is not the case when more resources are considered. Finally, iterating through all possible configurations and taking experimental samples for the exhaustive search is clearly infeasible for non-trivial combinations of resources and workloads.

These experiments and observations thus motivate us to design and implement a coordinated multi-resource partitioning algorithm based on an approximate system and application model, which we introduce next.

3 Dynamic Multi-Resource Allocation

In this section, we describe our approach to providing effective resource partitioning for database servers running on virtual storage. Our main objective is to meet an overall performance goal, e.g., minimize the overall latency, when running a set of database applications on a shared storage server. In order to achieve this, we use the following:

1. A performance model based on minimal statistics collection in order to approximate a *near*-optimal allocation of resources to applications according to our overall goal, and
2. An experimental sampling and statistical interpolation technique that refines the initial model.

In the following, we first introduce the problem statement, and an overview of our approach. Then, we introduce our performance model, and its sampling-based fine-tuning in detail.

3.1 Problem Statement

We study dynamic resource allocation to multiple applications in dynamic content servers with shared storage. In the most general case, let's assume that the system contains m resources and is hosting n applications. Our goal is to find the optimal configuration for partitioning

the m resources among the n applications. Let's denote with r_1, r_2, \dots, r_n the data access times of the n applications hosted by the service provider. For the purposes of this paper, we assume that the goal of the service provider is to minimize the sum of all data access latencies for all applications, i.e. $\mathcal{U} = \min \sum_{i=1}^n r_i$.

However, our approach does not depend on the particular goal we set. For example, alternatively, we can optimize the provider's revenue expressed as a *utility function* based on the application latencies. Whichever goal we set, we assume that our algorithm is aware of that goal, and can monitor application performance in order to compute the total benefit obtained for all applications, in any resource quota configuration.

Finding a practical solution to this problem is difficult, because the optimal resource allocation depends on many factors, including the (dynamic) access patterns of the applications, and how the inner mechanisms of each system component e.g., cache replacement policies, affect inter-dependencies between system resources.

3.2 Overview of Approach

Our technique determines per-application resource quotas in the database and storage caches, on the fly, in a transparent manner, with minimal changes to the DBMS, and no changes to existing interfaces between components. Towards this objective, we use an online performance estimation algorithm to dynamically determine the mapping between any given resource configuration setting and the corresponding application latency. While designing and implementing a performance model for guiding the resource partitioning search is non-trivial, our key insight is to design a model with sufficient expressiveness to incorporate i) tracking of dynamic access patterns, and ii) sufficiently generic assumptions about the inner mechanisms of the system components and the system as a whole.

For this purpose we collect a trace of I/O accesses at the DBMS buffer pool level and we use periodic sampling of the average disk latency for each application in a baseline configuration, where the application is given all the disk bandwidth. We feed the access trace and baseline disk latency for each application into a performance model, which computes the latency estimates for that application for all possible resource configurations. We thus obtain a set of resource-to-performance mapping functions, i.e., performance models, one for each application. Next, we enhance the accuracy of each performance model through experimental sampling. We use statistical regression to re-approximate the performance model by interpolating between the precomputed and experimentally gathered sample points.

We then use the corresponding per-application perfor-

mance models to determine the *near-optimal* allocation of resources to applications according to our overall goal. Specifically, we leverage the derived performance model of each application, and use *hill climbing* [21] to converge towards a partitioning setting that minimizes the combined application latencies. In the following subsection, we describe our model that estimates the performance of an application using multi-level caches and a shared disk.

3.3 Per-Application Performance Model

We use two key insights about the inner workings of the system, as explained next, to derive a close performance approximation, while at the same time reducing the complexity of the model as much as possible.

Key Assumptions and Ideas: The key assumptions we use about the system are i) that the cache replacement policy used in the cache hierarchy is known to be either the standard, uncoordinated LRU, or the coordinated DEMOTE [31] policy and ii) that the server is a closed-loop system i.e., it is interactive and the number of users is constant during periods of stable load. Both of these assumptions match our target system well, leading to a performance model with sufficient accuracy to find a near-optimal solution, as we will show in Section 6. With the assumptions above, our key idea is to replace the search space of a cache hierarchy with the simpler search space of a single level of cache, in order to obtain a close performance estimation, at higher speed, as described next.

3.3.1 Approximate Performance Model

We approximate the cache hierarchy with the model of a single-level cache, and we specialize this model for two most commonly deployed, or proposed cache replacement policies, i.e., uncoordinated LRU and coordinated DEMOTE [31]. We also derive a simplified disk model. Based on our models, assuming that the application is given quotas i.e., fractions ρ_c , ρ_s and ρ_d of the buffer pool cache, storage cache and disk bandwidth, respectively, we estimate the overall data access latency for the respective quotas through a combination of selective on-line measurements and computation.

In the following, we first introduce an approximation of the cache miss ratio of a two-level cache hierarchy, $\widehat{\mathcal{M}}(\rho_c, \rho_s)$, as a function of the cache quotas ρ_c and ρ_s , for the two types of replacement policies we consider. Then we introduce our disk model that computes the disk latency as a function of the disk quota, $L_d(\rho_d)$. Finally, we describe our overall data access latency model.

Modeling the Cache Hierarchy: In a cache hierarchy using the standard (uncoordinated) LRU replace-

ment policy at all levels, any cache miss from cache level q_i will result in bringing the needed block into all lower levels of the cache hierarchy, before providing the requested block to cache i . It follows that the block is *redundantly* cached at all cache levels, which is called the *inclusiveness* property [31]. Therefore, if an application is given a certain cache quota q_i at a level of cache i , any cache quotas q_j given at any lower level of cache j , with $q_j < q_i$ will be mostly wasteful.

In contrast, in a cache hierarchy using coordinated DEMOTE [31] cache replacement, when a block is fetched from disk, it is not kept in any lower cache levels. The lower cache levels cache blocks only when the block is evicted from a higher cache level. Therefore, the application benefits from the combined quotas at all levels due to cache *exclusiveness*. Based on these observations, we make the following simplifications to approximate the overall miss ratio of a two-level cache, i.e., $\widehat{\mathcal{M}}(\rho_c, \rho_s)$, based on a single-level cache model.

In an uncoordinated LRU cache hierarchy, only the maximum size quota given at any level of cache matters; therefore, we approximate the miss ratio of a two level cache, consisting of a buffer pool (with quota ρ_c) and a storage cache (with quota ρ_s) by the following formula:

$$\widehat{\mathcal{M}}(\rho_c, \rho_s) \approx \mathcal{M}_c(\max[\rho_c, \rho_s]) \quad (1)$$

In a coordinated DEMOTE cache hierarchy, the combined cache quotas given to the application at all levels of cache has the same effect on the overall miss ratio as giving the total quota in a single level of cache. Therefore, for DEMOTE cache replacement, we use the following formula to approximate the miss ratio of a two-level cache:

$$\widehat{\mathcal{M}}(\rho_c, \rho_s) \approx \mathcal{M}_c(\rho_c + \rho_s) \quad (2)$$

Modeling the Disk Latency: For modeling the disk latency, we observe that the typical server system is an *interactive*, closed-loop system. This means that, even if incoming load may vary over time, at any given point in time, the rate of serviced requests is roughly equal to the incoming request rate. According to the *interactive response time law* [10]:

$$L_d = \frac{N}{X} - z \quad (3)$$

where L_d is the response time of the storage server, including both I/O request scheduling and the disk access latency, N is the number of application threads, X is the throughput, and z is the think time of each application thread issuing requests to the disk.

We then use this formula to derive the average disk access latency for each application, when given a certain quota of the disk bandwidth. We assume that think time per thread is negligible compared to request processing time, i.e., we assume that I/O requests are arriving relatively frequently, and disk access time is significant. If this is not the case, the I/O component of a workload is likely not going to impact overall application performance. However, if necessary, more precision can be easily afforded e.g., by a *context tracking* approach, which allows the storage server to distinguish requests from different application threads [25], hence infer the average think time.

We further observe that the throughput of an application varies proportionally to the fraction of disk bandwidth that the application is given. Since disk saturation is unlikely in interactive environments with a limited number of I/O threads, this is very intuitive, but also verified through extensive validation experiments using a quanta-based scheduler and a variety of workloads.

Through a simple derivation, we arrive at the following formula:

$$L_d(\rho_d) = \frac{L_d(1)}{\rho_d} \quad (4)$$

where $L_d(1)$ is the *baseline disk latency* for an application, when the entire disk bandwidth is allocated to that application. This formula is intuitive. For example, if the entire disk was given to the application, i.e., $\rho_d = 1$, then the storage access latency is equal to the underlying disk access latency. On the other hand, if the application is given a small fraction of the disk bandwidth, i.e., $\rho_d \approx 0$, then the storage access latency is very high (approaches ∞).

Finally, the total cache quota allocated to an application influences the arrival rate of I/O requests at the disk, hence the *baseline disk latency* for that application. For example, a larger cache quota may result in a smaller disk queue, which in its turn limits opportunities for scheduling optimizations to minimize disk seeks. Hence, in the absence of disk bandwidth saturation, a larger cache quota may result in a higher *baseline disk latency* for the corresponding application.

Therefore, to compute the *baseline disk latency* for an application given a particular cache configuration, we use linear interpolation based on experimental measurements, taken for a few cache settings, instead of a single measurement.

Computing the Overall Performance Model: Assuming that the hit access latency in the buffer pool is negligible, the overall latency is determined by the accesses that miss in the buffer pool and either i) hit in the storage cache or ii) miss in the storage cache, hence access the disk.

Assuming that the access latency for a hit/miss in the storage cache is approximately the network/disk latency, i.e., L_{net}/L_d , respectively, then the average application latency is:

$$L_{avg}(\rho_c, \rho_a, \rho_d) = \mathcal{M}_c(\rho_c)\mathcal{H}_s(\rho_c, \rho_s)L_{net} + \mathcal{M}_c(\rho_c)\mathcal{M}_s(\rho_c, \rho_s)L_d(\rho_q) \quad (5)$$

where the miss (and hit) ratio at the storage cache, i.e., $\mathcal{M}_s(\rho_c, \rho_s)$, is a function of both the quota at the first level cache (ρ_c), and the quota at the second level cache (ρ_s), while the miss ratio of the buffer pool, $\mathcal{M}_c(\rho_c)$, is only a function of ρ_c . We can further approximate the fraction of accesses that miss in both levels of cache, hence reach the disk, i.e., $\mathcal{M}_c(\rho_c)\mathcal{M}_s(\rho_c, \rho_s)$ from the formula above, with the fraction of disk accesses given by the miss ratio of our previously introduced single-level cache model as:

$$\mathcal{M}_c(\rho_c)\mathcal{M}_s(\rho_c, \rho_s) = \widehat{\mathcal{M}}(\rho_c, \rho_s) \quad (6)$$

By using the previously derived models for $\widehat{\mathcal{M}}(\rho_c, \rho_s)$ e.g., in the case of uncoordinated LRU (Equation 1), we obtain:

$$\mathcal{M}_s(\rho_c, \rho_s) = \frac{\mathcal{M}_c(\max[\rho_c, \rho_s])}{\mathcal{M}_c(\rho_c)} \quad (7)$$

Therefore, we can approximate the miss ratio in the storage cache, $\mathcal{M}_s(\rho_c, \rho_s)$, in terms of the miss ratio of a single-level cache model. By replacing the respective miss/hit ratio of the storage cache in Equation 5, we derive the application latency based on our single-level cache performance model for either type of cache replacement policy.

Finally, in order to derive a complete resource-to-performance model, we perform access trace collection and compute the *miss ratio curve* (MRC) only at the *buffer pool* level. Then, we vary the quota allocations for the two caches and the disk bandwidth for the application, to all possible combinations *in the model*. For each quota setting, we then *compute* the corresponding application latencies based on the precomputed buffer pool MRC by Equation 5.

Model Adjustment to Dynamic Changes: The model needs periodic recalibration, in order to account for load variations. Recalibration involves taking new samples of the disk latency for each application in a few cache configurations, to recompute the *baseline disk latency*. A new application trace needs to be collected and the new MRC recomputed only if the application pattern changes. If a new application is co-scheduled on the

same infrastructure, we need to sample and compute the performance model only for the new application.

3.4 Sources of Inaccuracy

In our simple performance model we ignore the effects of *locking for concurrency control*, *dirty block flushes* for the cache model, and imperfect I/O isolation at *small disk quanta* for the disk model.

Specifically, whenever a dirty block evicted from the buffer pool is flushed to disk, the write access goes through all lower levels of cache on its way out. Hence, the evicted block remains cached in the storage cache, violating our assumption of *redundancy* for uncoordinated LRU caches, hence impacting cache miss ratio predictions.

Moreover, for low disk quanta, the disk scheduler incurs frequent and potentially large disk seeks between the data locations of different applications on disk. Thereby, our *disk latency* prediction, as well as the underlying I/O bandwidth isolation mechanism itself would be inaccurate in this case. In particular, the disk quanta cannot be less than the maximum duration of a disk read/write, which is that of a block size of 16KB in our case (for MySQL).

3.5 Model Fine-tuning

In order to fine-tune our performance model at run time, hence adaptively correct any inaccuracies, we use more expensive sampling-based approaches to correct the model at runtime. We collect experimental samples of application latency in various resource partitioning configurations, and use statistical regression i.e., *support vector machine regression* (SVR) [8], to re-approximate the resource-to-performance mapping function without sampling the search space exhaustively. SVR allows us to estimate the performance for configuration settings we haven't actuated, through interpolation between a given set of sample points.

We iteratively collect a set of k randomly selected sample points. Each sample represents the average application latency *measured* in a given configuration. We replace the respective points in our performance model with the new set of experimentally collected samples. Using *all* sample points, consisting of both computed and experimentally collected samples, we retrain the regression model. We also cross-validate the model by training the regression model on a sub-set of all samples and comparing with the regression function obtained using the remaining samples. If during cross-validation, we determine that the regression-based performance model is stable [8], then we conclude that we do not need to collect any more samples, and we have achieved a highly

accurate performance model for the respective application. Otherwise, we iterate through the above process until convergence is achieved.

3.6 Finding the Optimal Configuration

Based on the per-application performance models derived as above, we find the resource partitioning setting which gives the optimum i.e., lowest combined latency in our case, by using *hill climbing* with random-restarts [21]. The *hill climbing* algorithm is an iterative search algorithm that moves towards the direction of increasing combined utility value for all valid configurations at each iteration. To avoid reaching a local optimum, we conduct several searches from several points chosen randomly until each search reaches an optimum. We use the best result obtained from all searches.

4 Prototype Implementation

Our infrastructure (*Akash*¹) consists of a virtual storage system prototype designed to run on commodity hardware. It supports data accesses to multiple virtual volumes for any storage client, such as, database servers and file systems. It uses the Network Block Device (NBD) driver packaged with Linux to read and write logical blocks from the virtual storage system, as shown in Figure 3. NBD is a standard storage access protocol similar to iSCSI, supported by Linux. It provides a method to communicate with a storage server over the network. The client machine (shown in left) mounts the virtual volume as a NBD device (e.g., `/dev/nbd1`) which is used by MySQL as a raw disk partition, (e.g., `/dev/raw/raw1`). We modified existing *client* and *server* NBD protocol processing modules for the storage client and server, respectively, in order to interpose our storage cache and disk controller modules on the I/O communication path, as shown in the figure.

In addition, we provide interfaces for creating/destroying new virtual volumes and setting resource quanta per virtual volume. Our infrastructure supports a resource controller in charge of partitioning multiple levels of storage cache hierarchy and the storage bandwidth. The controller determines per-application resource quotas on the fly, based on our performance model introduced in Section 3, in a transparent manner, with minimal changes to the DBMS i.e., to collect access traces at the level of the buffer pool and to monitor performance. In addition, we modify the MySQL/InnoDB buffer pool to support dynamic partitioning and resizing of its buffer pool, since it does not currently provide these features.

¹*Akash* is a Sanskrit word meaning “sky” or “space”.

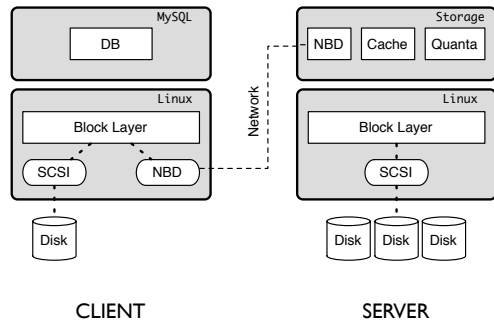


Figure 3: **Virtual Storage Architecture:** We show one client connected to a storage server using NBD.

4.1 Sampling Methodology

For each hosted application, and given configuration, in order to collect a sample point, we record the average and standard deviation of the data access latency, for the corresponding application in that configuration. For each sample point where we change the cache configuration, we wait for cache warm-up, until the application miss ratio is stable (which takes approximately 15 minutes on average in our experiments). Once the cache is stable, we monitor and record the application latency several times in order to reduce the noise in measurement. Once measured, sample points for an application can also be stored as an *application surface* on disk and later retrieved.

4.1.1 Efficient Sampling for Exhaustive Search

For the purpose of exhaustive sampling i.e., for comparing our model to measured optimum configurations (see Section 6.3.3), the controller iteratively sets the desired resource quotas and measures the application latency during each sampling period. We use the following rules of thumb in order to speed up the exhaustive sampling process:

Cost-aware Iteration: We sort resources in descending order of re-partitioning cost i.e., cache repartitioning has higher re-partitioning sampling cost compared to the disk due to the need to wait for cache warm-up in each new configuration. Therefore, we go through all cache partitioning possibilities as the outermost loop of our iterative exhaustive search; for each cache setting we go through all possible disk bandwidth settings in an inner loop, thus making fewer changes to stateful resources overall.

Order Reversal: The time to acquire a sample can be further reduced by iterating from larger cache quotas to smaller cache quotas i.e., from 1024MB to 32MB in a 1024MB cache. In this case, the cache warm-up of the

largest cache quota will be amortized over the sampling for all cache quotas for the application.

5 Evaluation

In this section, we describe several resource partitioning algorithms we use in our evaluation. In addition, we describe the benchmarks and methodology we use.

5.1 Algorithms used in Experiments

We compare our GLOBAL+ resource partitioning scheme, where we combine performance estimation and experimental sampling, with the following resource partitioning schemes.

1. GLOBAL: Is our resource allocation scheme where we use only the performance model. As opposed to the GLOBAL+ scheme, we do not add any runtime performance samples.
2. MRC: Uses MRC to perform cache partitioning *independently* at the buffer pool and the storage cache, based on access traces seen at that level. The disk bandwidth is equally divided among all applications.
3. DISK: Assigns equal portions of the cache to all applications at each level and explores all the possible configurations at the disk level.
4. MRC+DISK: Uses the cache configurations produced by the MRC scheme and then explores all the possible configurations for partitioning the disk bandwidth.
5. IDEAL+: Finds the configuration with best overall latency by exhaustive search through all possible cache and disk partitioning configurations. We allocate the caches in 64MB chunks, and the disk in 20ms quanta slices, yielding a total of $16 \times 16 \times 5 = 1280$ samples measured for each application. A more accurate solution can be obtained at finer grain increments, e.g., 32MB chunks, but the experiments are estimated to take months in this case.

5.2 Platform and Methodology

Our evaluation infrastructure consists of three machines: (1) a storage server running *Akash* to provide virtual disks, (2) a database server running MySQL, and (3) a load generator for the benchmarks.

We use three workloads: a simple micro-benchmark, called UNIFORM, and two industry-standard benchmarks, TPC-W and TPC-C. In our experiments, the benchmarks

share both the database and storage server machines, using the (default) LRU replacement, and containing 1GB of memory each. Cache quotas are allocated in 64MB increments, with a minimum of 64MB. Disk quotas are allocated as 20ms disk quanta slices.

We run our Web based applications (TPC-W) on a dynamic content infrastructure consisting of the Apache web server, the PHP application server and the MySQL/InnoDB (version 5.0.24) database engine. We run the Apache Web server and MySQL on Dell PowerEdge SC1450 with dual Intel Xeon processors running at 3.0 Ghz with 2GB of memory. MySQL connects to the raw device hosted by the NBD server. We run the NBD server on a Dell PowerEdge PE1950 with 8 Intel Xeon processors running at 2.8 Ghz with 3GB of memory. To maximize I/O bandwidth, we use RAID 0 on 15 10K RPM 250GB hard disks.

We configure *Akash* to use 16KB block size to match the MySQL/InnoDB block size. Each workload instance uses a different virtual volume: a 32GB virtual disk for TPC-C, a 64GB virtual disk for TPC-W, and a 64GB disk for UNIFORM. In addition, we use the Linux `O_DIRECT` mode to bypass any OS-level buffer caching and the `noop` I/O scheduler.

5.2.1 Benchmarks

UNIFORM: We generate the UNIFORM workload by accessing data in an uniformly random order. The behavior is controlled by two parameters: the size of the data set (d) and the memory working set size (w). We run the workload with $d=64GB$ and $w=1GB$.

TPC-W: The TPC-W benchmark from the Transaction Processing Council [1] is a transactional web benchmark designed for evaluating e-commerce systems. Several web interactions are used to simulate the activity of a retail store. The database size is determined by the number of items in the inventory and the size of the customer population. We use 100K items and 2.8 million customers which results in a database of about 4 GB. We use the *shopping* workload that consists of 20% writes. To fully stress our architecture, we run 10 TPC-W instances in parallel creating a database of 40 GB.

TPC-C: The TPC-C benchmark [20] simulates a whole-sale parts supplier that operates using a number of warehouse and sales districts. Each warehouse has 10 sales districts and each district serves 3000 customers. The workload involves transactions from a number of terminal operators centered around an order entry environment. There are 5 main transactions for: (1) entering orders (*New Order*), (2) delivering orders (*Delivery*), (3) recording payments (*Payment*), (4) checking the status of the orders (*Order Status*), and (5) monitoring the level of stock at the warehouses (*Stock Level*). Of the 5 transac-

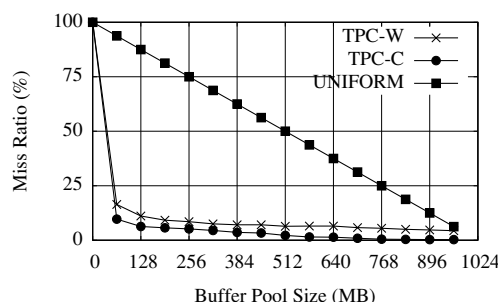


Figure 4: **Miss Ratio Curves:** At the buffer pool for our workloads.

tions, only *Stock Level* is read only, but constitutes only 4% of the workload mix. We scale TPC-C by using 128 warehouses, which gives a database footprint of 32GB.

6 Results

We evaluate our approach using the TPC-C and TPC-W industry standard benchmarks. We also use the synthetic UNIFORM workload. We first characterize our workloads by preliminary experiments showing their computed MRC at the buffer pool level, then report and compare the average data access latency, measured at the first level cache, for each application, when using different resource partitioning schemes.

6.1 Miss Ratio Curves

Figure 4 shows the *miss ratio curves* at the first level cache (buffer pool) for all applications. We can see that TPC-W and TPC-C are more cacheable than UNIFORM. UNIFORM has comparatively higher miss ratios, and it benefits greatly from larger cache allocations. On the other hand, TPC-W and TPC-C are less affected by cache allocations past 128MB.

6.2 Overall Performance

We run either identical workload instances, or different workload instances, concurrently, on our infrastructure, and compare the performance of our partitioning algorithms. Figures 5-8 show the latency of each application after each partitioner produces a solution. We also show the respective partitioning solutions, and the time in which they were achieved by each resource partitioner (we include the time to collect a reliable access trace in the timing for our algorithms, although this is overlapped with normal application execution).

We notice the following overall trends in our results. Our GLOBAL⁺ partitioner arrives at the same partition-

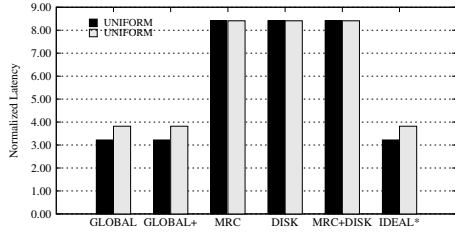


Figure 5: **Identical Instances:** Comparison for UNIFORM.

ing solution as, and provides identical performance to IDEAL*, at a fraction of the cost. The performance of the GLOBAL partitioner, based only on the computational model, is relatively close to the ideal performance as well. GLOBAL registers significant improvements with experimental sampling only for workload combinations that include TPC-C, an application with a substantial fraction of writes. Moreover, with one exception, our GLOBAL partitioner is both faster and generates better partitioning settings than the combination of single resource controllers i.e., the MRC+DISK partitioner.

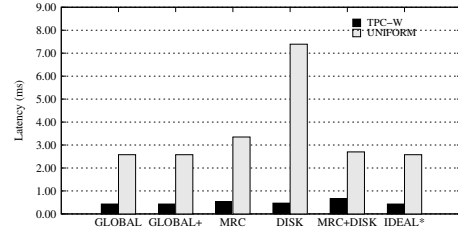
The single resource partitioning schemes, i.e., MRC and DISK, are limited in their ability to control performance. For example, DISK is ineffective for cache-bound workloads (see Figures 5, 6, 7). A more subtle point is that in some cases, the poor choices made by the MRC scheme can be corrected by providing more disk bandwidth to disadvantaged applications in the MRC+DISK scheme.

We discuss our performance results in detail next and we examine the accuracy of our model and its refinements in Section 6.3.

6.2.1 Identical Workload Instances

First, we look at cases where we run two instances of the same application. Figure 5 presents our results for the UNIFORM/UNIFORM configuration. The results for TPC-C/TPC-C and TPC-W/TPC-W are similar.

In these experiments, the *miss ratio curves* of the two applications are identical. Thus, the MRC/MRC+DISK/DISK schemes choose to partition the cache levels equally at both the client and storage caches. With this setting, due to cache inclusiveness, the second level cache, i.e., the storage cache, provides little benefit, resulting in poor performance for these partitioners. For the results shown in Figure 5, our GLOBAL scheme, finds a resource partitioning setting of 64MB/960MB and 960MB/64MB between the two instances of UNIFORM, at the buffer pool and storage caches respectively. This setting provides a much better cache usage scenario than equal partitioning of the two caches.

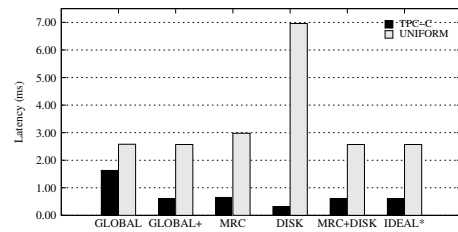


(a) Latency

Scheme	B.Pool		S.Cache		Quanta		Time (mins)
	TPC-W	UNIF	W	U	W	U	
GLOBAL	64	960	896	128	40	60	16
GLOBAL+	64	960	896	128	40	60	59
MRC	128	896	384	640	50	50	32
DISK	512	512	512	512	40	60	5
MRC+DISK	128	896	384	640	40	60	37
IDEAL*	64	960	896	128	40	60	3660

(b) Allocation

Figure 6: TPC-W/UNIFORM: Comparison for TPC-W (W) and UNIFORM (U) run concurrently.



(a) Latency

Scheme	B.Pool		S.Cache		Quanta		Time (mins)
	TPC-C	UNIF	C	U	C	U	
GLOBAL	64	960	896	128	40	60	16
GLOBAL+	64	960	512	512	40	60	760
MRC	128	896	512	512	50	50	32
DISK	512	512	512	512	40	60	5
MRC+DISK	128	896	512	512	40	60	37
IDEAL*	64	960	512	512	40	60	3660

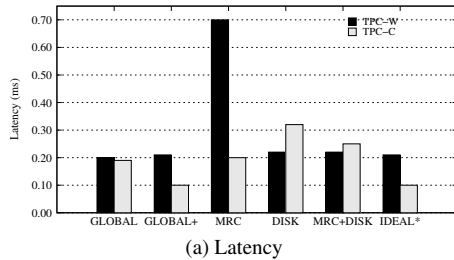
(b) Allocation

Figure 7: TPC-C/UNIFORM: Comparison for TPC-C (C) and UNIFORM (U) run concurrently.

Overall, GLOBAL provides the same partitioning solution as IDEAL* and obtains a factor of 2.4 speedup over MRC+DISK. For the experiments with two instances of TPC-W and TPC-C, GLOBAL obtains a factor of 1.05 and 1.5 speedup, respectively, over MRC+DISK.

6.2.2 Different Workload Instances

Figures 6-8 present our results for different concurrent workloads. The results show that the allocations chosen by the GLOBAL partitioner are non-trivial, and good



(a) Latency

Scheme	B.Pool		S.Cache		Quanta		Time (mins)
	TPC-W	TPC-C	W	C	W	C	
GLOBAL	192	960	896	128	60	40	16
GLOBAL ⁺	256	768	768	256	60	40	760
MRC	384	640	384	640	50	50	32
DISK	512	512	512	512	50	50	5
MRC+DISK	384	640	384	640	60	40	37
IDEAL [*]	256	768	768	256	60	40	3660

(b) Allocation

Figure 8: TPC-W/TPC-C: Comparison for TPC-W (W) and TPC-C (C) run concurrently.

performance is obtained only when the settings of all resources are considered.

First, we examine the TPC-W/UNIFORM configuration, shown in Figure 6. The UNIFORM workload has both larger cache and disk requirements than TPC-W. Since the *miss ratio curve* of UNIFORM is steeper than that of TPC-W, once the first 128MB is allocated to TPC-W, the MRC partitioner allocates the rest of the buffer pool (896MB) to UNIFORM. However, UNIFORM is penalized by the 50/50 disk bandwidth partitioning in this case. On the other hand, the DISK partitioner selects a 60/40 disk bandwidth allocation in favor of UNIFORM. But, dividing the caches 50/50 results in poor performance for this partitioner. The MRC+DISK scheme corrects the disk quanta allocation of the MRC scheme. However, due to the underlying uncoordinated LRU replacement policy, it fails to obtain a synergistic configuration for the two caches. Therefore, GLOBAL performs a factor of 1.12 better than MRC+DISK, by obtaining a better cache configuration overall, in addition to allocating the disk bandwidth in favor of UNIFORM. GLOBAL performs a factor of 1.29 better than MRC, and a factor of 2.61 better than DISK.

Next, we look at the TPC-C/UNIFORM configuration, shown in Figure 7. The results are similar to the TPC-W/UNIFORM configuration, with one exception. The model for our GLOBAL partitioner mispredicts the cache behavior at the storage cache. The assumption about block redundancy between the buffer pool and storage cache does not hold for TPC-C, an application with a substantial fraction of writes. Hence, allocating more storage cache to TPC-C, as in the solutions of all other par-

tioners is beneficial, resulting in increased hit rates in this cache. The DISK and MRC partitioners under-perform for the same reason as before i.e., because allocating either cache or disk resources 50/50 penalizes UNIFORM. Hence, GLOBAL⁺ performs a factor of 1.14, and 2.29 better than MRC, and DISK, respectively, and similar to MRC+DISK.

Finally, we study the TPC-W/TPC-C configuration, shown in Figure 8. As the *miss ratio curve* for TPC-C is slightly steeper than TPC-W, the MRC partitioner allocates a larger fraction of the buffer pool (640MB) to TPC-C. Moreover, the *miss ratio curves* of the two applications are similar to each other at the storage cache level. Therefore, the same greedy MRC cache algorithm allocates a larger fraction of the storage cache (640MB) to TPC-C as well. This results in over-allocation of total cache space to TPC-C, severely penalizing TPC-W, when compared to the cache configuration, and performance achieved by IDEAL^{*} (and our GLOBAL⁺). Allocating a larger disk fraction to TPC-W in MRC+DISK compensates for the poor cache partitioning of MRC alone. The GLOBAL⁺ scheme allocates a larger proportion of the storage cache than GLOBAL to TPC-C, correcting the initial mis-prediction, while still balancing the allocation at the two caches for avoiding redundancy, hence providing overall better performance. As a result, GLOBAL⁺ is a factor of 2.89 better than MRC, a factor of 1.72 better than DISK, and a factor of 1.51 better than MRC+DISK.

6.3 Performance Model Accuracy

In this section, we evaluate the accuracy of our cache and disk approximations in our performance model. In addition, we present results for online refinement of our model through experimental sampling.

6.3.1 Two-level Cache Approximation

We evaluate the accuracy of the two-level cache miss ratio prediction. Figure 9 presents our results for TPC-W and TPC-C. We first provide a detailed analysis for TPC-W, for three buffer pool size (64MB, 256MB, 512MB) and a range of storage cache sizes, where we plot two cache *miss ratio curves*: *experimentally measured* (solid lines) and *predicted by model* (dashed lines). As we can see, the predicted and measured *miss ratio curves* are close together, hence, our cache approximation is accurate in calculating the miss ratio at the storage cache. The areas of inaccuracy, where the relative error is greater than 2%, occur when the storage cache is equal to the buffer pool size i.e., 512M. The replacement policy is affected by concurrency control i.e., through the fix/unfix of buffer blocks and some other thread optimizations to mitigate cache pollution for table scans, in this case.

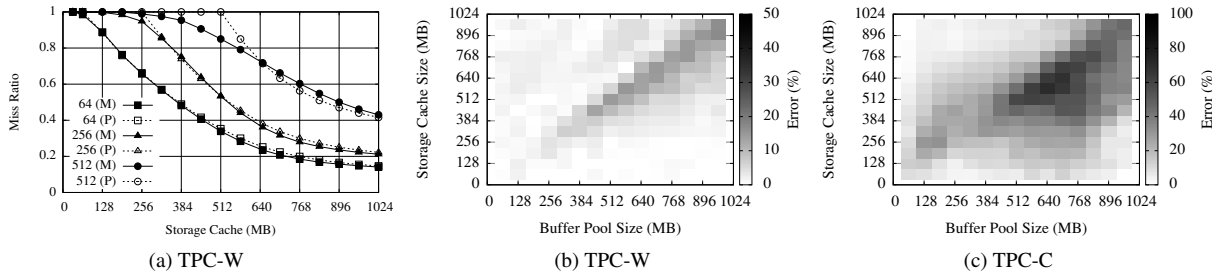


Figure 9: **Two-level Cache Approximation:** Errors for cache configurations with TPC-W and TPC-C. Figure 9a shows the Measured and Predicted miss ratio curves for buffer pool sizes (64MB,256MB,512MB). Figures 9b-9c show error *heatmaps* where light/dark colors represent low/high error, respectively. The magnitude of the error is shown in the legend on the right.

We further present the error of our model as a more general *heat-map*, where low errors (0-20%) are shown in light colors, whereas higher errors are shown in darker colors, for a wide range of cache configurations, for both our benchmarks. For both benchmarks, the area of any significant inaccuracy is where the two cache sizes are equal, especially for large cache sizes. However, these very configurations are unlikely to be used as an allocation solution, because they correspond to a high level of redundancy for uncoordinated two-level LRU caches. Moreover, for high cache sizes, the miss ratio of most applications is low, hence the error is less relevant. The errors are higher for TPC-C due to its large fraction of writes, hence unpredictable hits in the storage cache for *dirty blocks* previously evicted from the buffer pool. For both benchmarks, the error falls below 2% when the storage cache is at least a factor of 2 larger than the buffer pool size.

6.3.2 Quanta-based Scheduler Approximation

We evaluate the accuracy of our disk latency approximation, when using a quanta-based scheduler (Equation 4). We plot both the predicted and the measured disk latency, for each application, by varying the storage bandwidth quanta. Figure 10a and Figure 10b present our results for TPC-C and TPC-W, respectively. In each graph, we plot and compare two lines: *measured* (solid lines) and *predicted* (dashed lines), for different cache sizes (given mostly at the buffer pool).

Overall, the predicted disk latency significantly deviates from the measured latency only for small quanta values. Moreover, slightly higher errors can be observed for higher cache sizes. In both of these cases, the explanation is the higher variability of the average disk latency over time when i) the underlying disk bandwidth isolation is less effective due to frequent switching between workloads and ii) disk scheduling optimizations are less ef-

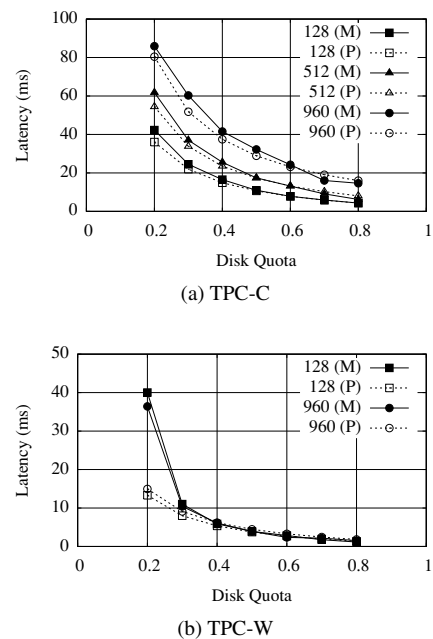


Figure 10: **Accuracy of Quanta Scheduler Approximation:** We plot the Predicted and Measured disk latency by varying the disk scheduler quota, in different cache configurations, from 128MB cache to 960MB cache.

fective and reliable due to fewer requests in the scheduler queue. Moreover, our model ignores the “think time” between successive requests for the same workload. On the other hand, we can see that our model successfully captures the latency deviations due to changes in the cache size for TPC-C.

6.3.3 Model Refinement with Runtime Sampling

As shown, our model is inaccurate in very localized areas of the total search space, where inaccuracies may not

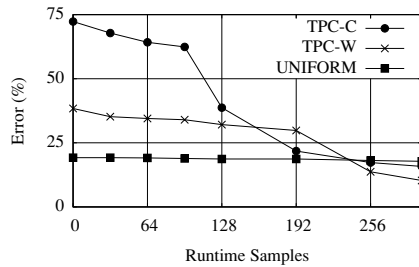


Figure 11: **Online Sampling:** We refine our model accuracy at runtime with experimental sampling.

matter, or can be improved by experimental sampling. Figure 11 shows the accuracy improvement through online performance sampling. In the x -axis we show the number of samples added to our performance model experimentally, and on the y -axis we show the error between the predicted and the actual latencies. For both TPC-W and TPC-C, adding samples by online sampling significantly reduces the error rate from 72% to 16% for TPC-C and from 38% to 10% for TPC-W.

7 Related Work

Previous related work has focused on dynamic allocation and/or controlling either memory allocation or disk bandwidth partitioning among competing workloads.

Dynamic Memory Partitioning: Dynamic memory allocation algorithms have been studied in the VMWare ESX server [28]. The algorithm estimates the *working-set* sizes of each VM and periodically adjusts each VM's memory allocation such that performance goals are met. Adaptive cache management based on application patterns or query classes has been extensively studied in database systems. For example, the DBMIN algorithm [7] uses the knowledge of the various patterns of queries to allocate buffer pool memory efficiently. In addition, many cache replacement algorithms have been studied e.g., LRU-k [15], in the presence of concurrent workloads. LRU-k prevents useful buffer pages from being evicted due to sequential scans running concurrently. Brown et al. [3] study schemes to ensure per-class response time goals in a system executing queries of multiple classes by sizing the different memory regions. Finally, recently, IBM DB2 added the self-tuning memory manager (STMM) to size different memory regions [26].

Disk Bandwidth Partitioning: Dynamic allocation of the disk bandwidth has been studied to provide QoS at the storage server. Just like in our prototype, SLEDs [5], Façade [12], SFQ [11], and Argon [27] place a scheduling tier above the existing disk scheduler in order to control the I/Os issued to the underlying disk. However,

these techniques assume that proportions are known e.g., set manually. However, more recent techniques, e.g., Cello [22], YFQ [4] and Fahrard [19] build QoS-aware disk schedulers, which make low-level scheduling decisions that strive to minimize seek times as well as maintain quality of service.

Multi-resource Partitioning: Multi-resource partitioning is an emerging area of research where multiple resources are partitioned to provide isolation and QoS for several competing applications. Wachs et al. [27] show the benefit of considering both cache allocation and disk bandwidth allocation to improve the performance in shared storage servers. However, the resource allocation is done after modelling applications through extensive profiling. Chanda et al. [6] implement priority scheduling at the web and database server levels. Wang et al. [30] extend the SFQ [11] algorithm to several storage servers. Padala et al. [17] study methods to allocate memory and CPU to several virtual machines located within the same physical server. However, these papers focus on either i) dynamic partitioning and/or quota enforcement of a single resource on multiple machines [6, 30] or ii) allocation of multiple resources within a single machine [17, 27]. In our study, we have shown that global resource partitioning of multiple resources located at different tiers results in significant performance gains.

8 Conclusions

Resource allocation to applications on the fly is increasingly desirable in shared data centers with server consolidation. While many techniques for enforcing a known allocation exist, dynamically finding the appropriate per-resource application quotas has received less attention. The challenge is the exponential growth of the search space for the optimal solution with the number of applications and resources. Hence, exhaustively evaluating application performance for all possible configurations experimentally is infeasible.

Our contribution is an effective multi-resource allocation technique based on a unified resource-to-performance model incorporating i) pre-existing generic knowledge about the system and inter-dependencies between system resources e.g., due to cache replacement policies and ii) application access tracking and baseline system metrics captured on-line.

We show through experiments using several standard e-commerce benchmarks and synthetic workloads that our performance model is sufficiently accurate in order to converge towards a near-optimal global partitioning solution within minutes. At the same time, our performance model effectively optimizes high-level performance goals, providing up to factors of 2.9 and 2.4 im-

provement compared to state-of-the-art single-resource controllers, and their ad-hoc combination, respectively.

Acknowledgments

We are grateful to our shepherd, David Black, and the anonymous reviewers for their valuable and detailed feedback on our paper. We also thank the systems group at University of Toronto for their excellent suggestions and comments. Finally, we acknowledge the generous support of our sponsors: Natural Sciences and Engineering Research Council (NSERC), Ontario Centers of Excellence (OCE), Ontario Ministry of Research and Innovation, IBM Center of Advanced Studies, IBM Research, Intel and Bell Canada. Gokul Soundararajan is supported by an NSERC Canada Graduate Scholarship.

References

- [1] Transaction processing council. <http://www.tpc.org>.
- [2] BARHAM, P. T., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T. L., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *SOSP* (2003), pp. 164–177.
- [3] BROWN, K. P., CAREY, M. J., AND LIVNY, M. Managing Memory to Meet Multiclass Workload Response Time Goals. In *VLDB* (1993), pp. 328–341.
- [4] BRUNO, J. L., BRUSTOLONI, J. C., GABBER, E., ÖZDEN, B., AND SILBERSCHATZ, A. Disk Scheduling with Quality of Service Guarantees. In *ICMCS, Vol. 2* (1999), pp. 400–405.
- [5] CHAMBLISS, D. D., ALVAREZ, G. A., PANDEY, P., JADAV, D., XU, J., MENON, R., AND LEE, T. P. Performance Virtualization for Large-Scale Storage Systems. In *SRDS* (2003), pp. 109–118.
- [6] CHANDA, A., ELMEELEGGY, K., COX, A. L., AND ZWAENEPOEL, W. Causeway: Support for Controlling and Analyzing the Execution of Multi-tier Applications. In *Middleware* (2005), pp. 42–59.
- [7] CHOU, H.-T., AND DEWITT, D. J. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *VLDB* (Stockholm, Sweden, August 1985), pp. 127–141.
- [8] DRUCKER, H., BURGESS, C. J. C., KAUFMAN, L., SMOLA, A. J., AND VAPNIK, V. Support Vector Regression Machines. In *NIPS* (1996), pp. 155–161.
- [9] GULATI, A., MERCHANT, A., AND VARMAN, P. J. pClock: an arrival curve based approach for QoS guarantees in shared storage systems. *SIGMETRICS* 35, 1 (2007), 13–24.
- [10] JAIN, R. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modelling*. John Wiley & Sons, New York, 1991.
- [11] JIN, W., CHASE, J. S., AND KAUR, J. Interposed Proportional Sharing for a Storage Service Utility. In *SIGMETRICS* (2004), pp. 37–48.
- [12] LUMB, C. R., MERCHANT, A., AND ALVAREZ, G. A. Façade: Virtual storage devices with performance guarantees. In *FAST* (Berkeley, CA, USA, 2003), USENIX Association, pp. 131–144.
- [13] MATTSON, R., GECSEI, J., SLUTZ, D., AND TRAIGER, I. Evaluation techniques for storage hierarchies. In *IBM System Journal* (1970), pp. 78–117.
- [14] MOGUL, J. C. Emergent (mis)behavior vs. complex software systems. In *EuroSys* (2006), pp. 293–304.
- [15] O’NEIL, E. J., O’NEIL, P. E., AND WEIKUM, G. The lru-k page replacement algorithm for database disk buffering. In *SIGMOD* (1993), pp. 297–306.
- [16] OZMEN, O., SALEM, K., UYSAL, M., AND ATTAR, M. H. S. Storage Workload Estimation for Database Management Systems. In *SIGMOD* (2007), pp. 377–388.
- [17] PADALA, P., SHIN, K. G., ZHU, X., UYSAL, M., WANG, Z., SINGHAL, S., MERCHANT, A., AND SALEM, K. Adaptive control of virtualized resources in utility computing environments. In *EuroSys* (2007), pp. 289–302.
- [18] POPESCU, A., AND GHANBARI, S. A Study on Performance Isolation Approaches for Consolidated Storage. *Technical Report, University of Toronto* (May 2008).
- [19] POVZNER, A., KALDEWEY, T., BRANDT, S. A., GOLDING, R. A., WONG, T. M., AND MALTZAHN, C. Efficient Guaranteed Disk Request Scheduling with Fahrrad. In *EuroSys* (2008), pp. 13–25.
- [20] RAAB, F. TPC-C-The Standard Benchmark for Online transaction Processing (OLTP). In *The Benchmark Handbook*. 1993.
- [21] RUSSELL, S. J., NORVIG, P., CANDY, J. F., MALIK, J. M., AND EDWARDS, D. D. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [22] SHENOY, P. J., AND VIN, H. M. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. *SIGMETRICS* 26, 1 (1998), 44–55.
- [23] SOUNDARARAJAN, G., AND AMZA, C. Towards End-to-End Quality of Service: Controlling I/O Interference in Shared Storage Servers. In *Middleware* (2008), pp. 287–305.
- [24] SOUNDARARAJAN, G., CHEN, J., SHARAF, M. A., AND AMZA, C. Dynamic Partitioning of the Cache Hierarchy in Shared Data Centers. *PVLDB* 1, 1 (2008), 635–646.
- [25] SOUNDARARAJAN, G., MIHAILESCU, M., AND AMZA, C. Context Aware Block Prefetching at the Storage Server. In *USENIX* (2008), pp. 377–390.
- [26] STORM, A. J., GARCIA-ARELLANO, C., LIGHTSTONE, S., DIAO, Y., AND SURENDRA, M. Adaptive Self-tuning Memory in DB2. In *VLDB* (2006), pp. 1081–1092.
- [27] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: Performance Insulation for Shared Storage Servers. In *FAST* (Berkeley, CA, USA, 2007), USENIX Association, pp. 5–5.
- [28] WALDSPURGER, C. A. Memory Resource Management in VMware ESX Server. In *OSDI* (2002), pp. 181–194.
- [29] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *OSDI* (1994), pp. 1–11.
- [30] WANG, Y., AND MERCHANT, A. Proportional-share Scheduling for Distributed Storage Systems. In *FAST* (Berkeley, CA, USA, 2007), USENIX Association, pp. 4–4.
- [31] WONG, T. M., AND WILKES, J. My Cache or Yours? Making Storage More Exclusive. In *USENIX* (2002), pp. 161–175.
- [32] ZHOU, P., PANDEY, V., SUNDARESAN, J., RAGHURAMAN, A., ZHOU, Y., AND KUMAR, S. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *ASPLOS* (2004), pp. 177–188.

PARDA: Proportional Allocation of Resources for Distributed Storage Access

Ajay Gulati Irfan Ahmad Carl A. Waldspurger

VMware Inc.

{agulati,irfan,carl}@vmware.com

Abstract

Rapid adoption of virtualization technologies has led to increased utilization of physical resources, which are multiplexed among numerous workloads with varying demands and importance. Virtualization has also accelerated the deployment of shared storage systems, which offer many advantages in such environments. Effective resource management for shared storage systems is challenging, even in research systems with complete end-to-end control over all system components. Commercially-available storage arrays typically offer only limited, proprietary support for controlling service rates, which is insufficient for isolating workloads sharing the same storage volume or LUN.

To address these issues, we introduce PARDA, a novel software system that enforces proportional-share fairness among distributed hosts accessing a storage array, without assuming any support from the array itself. PARDA uses latency measurements to detect overload, and adjusts issue queue lengths to provide fairness, similar to aspects of flow control in FAST TCP. We present the design and implementation of PARDA in the context of VMware ESX Server, a hypervisor-based virtualization system, and show how it can be used to provide differential quality of service for unmodified virtual machines while maintaining high efficiency. We evaluate the effectiveness of our implementation using quantitative experiments, demonstrating that this approach is practical.

1 Introduction

Storage arrays form the backbone of modern data centers by providing consolidated data access to multiple applications simultaneously. Deployments of consolidated storage using Storage Area Network (SAN) or Network-Attached Storage (NAS) hardware are increasing, motivated by easy access to data from anywhere at any time, ease of backup, flexibility in provisioning, and centralized administration. This trend is further fueled by the proliferation of virtualization technologies, which rely on shared storage to support features such as live migration of workloads across hosts.

A typical virtualized data center consists of multiple physical hosts, each running several virtual machines

(VMs). Many VMs may compete for access to one or more logical units (LUNs) on a single storage array. The resulting contention at the array for resources such as controllers, caches, and disk arms leads to unpredictable IO completion times. Resource management mechanisms and policies are required to enable performance isolation, control service rates, and enforce service-level agreements.

In this paper, we target the problem of providing coarse-grained fairness to VMs, without assuming any support from the storage array itself. We also strive to remain work-conserving, so that the array is utilized efficiently. We focus on proportionate allocation of IO resources as a flexible building block for constructing higher-level policies. This problem is challenging for several reasons, including the need to treat the array as an unmodifiable black box, unpredictable array performance, uncertain available bandwidth, and the desire for a scalable decentralized solution.

Many existing approaches [13, 14, 16, 21, 25, 27, 28] allocate bandwidth among multiple applications running on a single host. In such systems, one centralized scheduler has complete control over all requests to the storage system. Other centralized schemes [19, 30] attempt to control the queue length at the device to provide tight latency bounds. Although centralized schedulers are useful for host-level IO scheduling, in our virtualized environment we need an approach for coordinating IO scheduling across multiple independent hosts accessing a shared storage array.

More decentralized approaches, such as Triage [18], have been proposed, but still rely on centralized measurement and control. A central agent adjusts per-host bandwidth caps over successive time periods and communicates them to hosts. Throttling hosts using caps can lead to substantial inefficiency by under-utilizing array resources. In addition, host-level changes such as VMs becoming idle need to propagate to the central controller, which may cause a prohibitive increase in communication costs.

We instead map the problem of distributed storage access from multiple hosts to the problem of flow control in networks. In principle, fairly allocating storage bandwidth with high utilization is analogous to distributed hosts trying to estimate available network bandwidth and consuming it in a fair manner. The network is effectively a black box to the hosts, providing little or no information about its current

state and the number of participants. Starting with this loose analogy, we designed PARDA, a new software system that enforces coarse-grained proportional-share fairness among hosts accessing a storage array, while still maintaining high array utilization.

PARDA uses the IO latency observed by each host as an indicator of load at the array, and uses a control equation to adjust the number of IOs issued per host, *i.e.*, the host *window size*. We found that variability in IO latency, due to both request characteristics (*e.g.*, degree of sequentiality, reads vs. writes, and IO size) and array internals (*e.g.*, request scheduling, caching and block placement) could be magnified by the independent control loops running at each host, resulting in undesirable divergent behavior.

To handle such variability, we found that using the average latency observed across *all* hosts as an indicator of overall load produced stable results. Although this approach does require communication between hosts, we need only compute a simple average for a single metric, which can be accomplished using a lightweight, decentralized aggregation mechanism. PARDA also handles idle VMs and bursty workloads by adapting per-host weights based on long-term idling behavior, and by using a local scheduler at the host to handle short-term bursts. Integrating with a local proportional-share scheduler [10] enables fair end-to-end access to VMs in a distributed environment.

We implemented a complete PARDA prototype in the VMware ESX Server hypervisor [24]. For simplicity, we assume all hosts use the same PARDA protocol to ensure fairness, a reasonable assumption in most virtualized clusters. Since hosts run compatible hypervisors, PARDA can be incorporated into the virtualization layer, and remain transparent to the operating systems and applications running within VMs. We show that PARDA can maintain cluster-level latency close to a specified threshold, provide coarse-grained fairness to hosts in proportion to per-host weights, and provide end-to-end storage IO isolation to VMs or applications while handling diverse workloads.

The next section presents our system model and goals in more detail. Section 3 develops the analogy to network flow control, and introduces our core algorithm, along with extensions for handling bursty workloads. Storage-specific challenges that required extensions beyond network flow control are examined in Section 4. Section 5 evaluates our implementation using a variety of quantitative experiments. Related work is discussed in section 6, while conclusions and directions for future work are presented in Section 7.

2 System Model

PARDA was designed for distributed systems such as the one shown in Figure 1. Multiple hosts access one or more storage arrays connected over a SAN. Disks in storage ar-

rays are partitioned into RAID groups, which are used to construct LUNs. Each LUN is visible as a storage device to hosts and exports a cluster filesystem for distributed access. A VM disk is represented by a file on one of the shared LUNs, accessible from multiple hosts. This facilitates migration of VMs between hosts, avoiding the need to transfer disk state.

Since each host runs multiple virtual machines, the IO traffic issued by a host is the aggregated traffic of all its VMs that are currently performing IO. Each host maintains a set of pending IOs at the array, represented by an *issue queue*. This queue represents the IOs scheduled by the host and currently pending at the array; additional requests may be pending at the host, waiting to be issued to the storage array. Issue queues are typically per-LUN and have a fixed maximum *issue queue length*¹ (*e.g.*, 64 IOs per LUN).

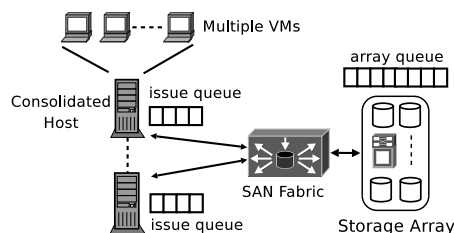


Figure 1: Storage array accessed by distributed hosts/VMs.

IO requests from multiple hosts compete for shared resources at the storage array, such as controllers, cache, interconnects, and disks. As a result, workloads running on one host can adversely impact the performance of workloads on other hosts. To support performance isolation, resource management mechanisms are required to specify and control service rates under contention.

Resource allocations are specified by numeric *shares*, which are assigned to VMs that consume IO resources.² A VM is entitled to consume storage array resources proportional to its share allocation, which specifies the relative importance of its IO requests compared to other VMs. The IO shares associated with a host is simply the total number of per-VM shares summed across all of its VMs. Proportional-share fairness is defined as providing storage array service to hosts in proportion to their shares.

In order to motivate the problem of IO scheduling across multiple hosts, consider a simple example with four hosts running a total of six VMs, all accessing a common shared LUN over a SAN. Hosts 1 and 2 each run two Linux VMs configured with OLTP workloads using Filebench [20].

¹The terms *queue length*, *queue depth*, and *queue size* are used interchangeably in the literature. In this paper, we will also use the term *window size*, which is common in the networking literature.

²Shares are alternatively referred to as *weights* in the literature. Although we use the term *VM* to be concrete, the same proportional-share framework can accommodate other abstractions of resource consumers, such as applications, processes, users, or groups.

Host	VM Types	s_1, s_2	VM1	VM2	T_h
1	2×OLTP	20, 10	823 Ops/s	413 Ops/s	1240
2	2×OLTP	10, 10	635 Ops/s	635 Ops/s	1250
3	1×Micro	20	710 IOPS	n/a	710
4	1×Micro	10	730 IOPS	n/a	730

Table 1: Local scheduling does not achieve inter-host fairness. Four hosts running six VMs without PARDA. Hosts 1 and 2 each run two OLTP VMs, and hosts 3 and 4 each run one micro-benchmark VM issuing 16 KB random reads. Configured shares (s_i), Filebench operations per second (Ops/s), and IOPS (T_h for hosts) are respected within each host, but not across hosts.

Hosts 3 and 4 each run a Windows Server 2003 VM with Iometer [1], configured to generate 16 KB random reads. Table 1 shows that the VMs are configured with different share values, entitling them to consume different amounts of IO resources. Although a local start-time fair queuing (SFQ) scheduler [16] does provide proportionate fairness within each individual host, per-host local schedulers alone are insufficient to provide isolation and proportionate fairness *across* hosts. For example, note that the aggregate throughput (in IOPS) for hosts 1 and 2 is quite similar, despite their different aggregate share allocations. Similarly, the Iometer VMs on hosts 3 and 4 achieve almost equal performance, violating their specified 2 : 1 share ratio.

Many units of allocation have been proposed for sharing IO resources, such as Bytes/s, IOPS, and disk service time. Using Bytes/s or IOPS can unfairly penalize workloads with large or sequential IOs, since the cost of servicing an IO depends on its size and location. Service times are difficult to measure for large storage arrays that service hundreds of IOs concurrently.

In our approach, we conceptually partition the array queue among hosts in proportion to their shares. Thus two hosts with equal shares will have equal queue lengths, but may observe different throughput in terms of Bytes/s or IOPS. This is due to differences in per-IO cost and scheduling decisions made within the array, which may process requests in the order it deems most efficient to maximize aggregate throughput. Conceptually, this effect is similar to that encountered when time-multiplexing a CPU among various workloads. Although workloads may receive equal time slices, they will retire different numbers of instructions due to differences in cache locality and instruction-level parallelism. The same applies to memory and other resources, where equal hardware-level allocations do not necessarily imply equal application-level progress.

Although we focus on issue queue slots as our primary fairness metric, each queue slot could alternatively represent a fixed-size IO operation (*e.g.*, 16 KB), thereby providing throughput fairness expressed in Bytes/s. However, a key benefit of managing queue length instead of throughput is that it automatically compensates workloads with lower

per-IO costs at the array by allowing them to issue more requests. By considering the actual cost of the work performed by the array, overall efficiency remains higher.

Since there is no central server or proxy performing IO scheduling, and no support for fairness in the array, a per-host *flow control* mechanism is needed to enforce specified resource allocations. Ideally, this mechanism should achieve the following goals: (1) provide coarse-grained proportional-share fairness among hosts, (2) maintain high utilization, (3) exhibit low overhead in terms of per-host computation and inter-host communication, and (4) control the overall latency observed by the hosts in the cluster.

To meet these goals, the flow control mechanism must determine the maximum number of IOs that a host can keep pending at the array. A naive method, such as using static per-host issue queue lengths proportional to each host’s IO shares, may provide reasonable isolation, but would not be work-conserving, leading to poor utilization in underloaded scenarios. Using larger static issue queues could improve utilization, but would increase latency and degrade fairness in overloaded scenarios.

This tradeoff between fairness and utilization suggests the need for a more dynamic approach, where issue queue lengths are varied based on the current level of contention at the array. In general, queue lengths should be increased under low contention for work conservation, and decreased under high contention for fairness. In an equilibrium state, the queue lengths should converge to different values for each host based on their share allocations, so that hosts achieve proportional fairness in the presence of contention.

3 IO Resource Management

In this section we first present the analogy between flow control in networks and distributed storage access. We then explain our control algorithm for providing host-level fairness, and discuss VM-level fairness by combining cluster-level PARDA flow control with local IO scheduling at hosts.

3.1 Analogy to TCP

Our general approach maps the problem of distributed storage management to flow control in networks. TCP running at a host implements flow control based on two signals from the network: round trip time (RTT) and packet loss probability. RTT is essentially the same as IO request latency observed by the IO scheduler, so this signal can be used without modification.

However, there is no useful analog of network packet loss in storage systems. While networking applications expect dropped packets and handle them using retransmission, typical storage applications do not expect dropped IO requests, which are rare enough to be treated as hard failures.

Thus, we use IO latency as our only indicator of congestion at the array. To detect congestion, we must be able to distinguish underloaded and overloaded states. This is accomplished by introducing a *latency threshold* parameter, denoted by \mathcal{L} . Observed latencies greater than \mathcal{L} may trigger a reduction in queue length. FAST TCP, a recently-proposed variant of TCP, uses packet latency instead of packet loss probability, because loss probability is difficult to estimate accurately in networks with high bandwidth-delay products [15]. This feature also helps in high-bandwidth SANs, where packet loss is unlikely and TCP-like AIMD (additive increase multiplicative decrease) mechanisms can cause inefficiencies. We use a similar adaptive approach based on average latency to detect congestion at the array.

Other networking proposals such as RED [9] are based on early detection of congestion using information from routers, before a packet is lost. In networks, this has the added advantage of avoiding retransmissions. However, most proposed networking techniques that require router support have not been adopted widely, due to overhead and complexity concerns; this is analogous to the limited QoS support in current storage arrays.

3.2 PARDA Control Algorithm

The PARDA algorithm detects overload at the array based on average IO latency measured over a fixed time period, and adjusts the host's issue queue length (*i.e.*, window size) in response. A separate instance of the PARDA control algorithm executes on each host.

There are two main components: latency estimation and window size computation. For latency estimation, each host maintains an exponentially-weighted moving average of IO latency at time t , denoted by $L(t)$, to smooth out short-term variations. The weight given to past values is determined by a smoothing parameter $\alpha \in [0, 1]$. Given a new latency observation l ,

$$L(t) = (1 - \alpha) \times l + \alpha \times L(t - 1) \quad (1)$$

The window size computation uses a control mechanism shown to exhibit stable behavior for FAST TCP:

$$w(t + 1) = (1 - \gamma)w(t) + \gamma \left(\frac{\mathcal{L}}{L(t)} w(t) + \beta \right) \quad (2)$$

Here $w(t)$ denotes the window size at time t , $\gamma \in [0, 1]$ is a smoothing parameter, \mathcal{L} is the system-wide latency threshold, and β is a per-host parameter that reflects its IO shares allocation.

Whenever the average latency $L > \mathcal{L}$, PARDA decreases the window size. When the overload subsides and $L < \mathcal{L}$, PARDA increases the window size. Window size adjustments are based on latency measurements, which indicate

load at the array, as well as per-host β values, which specify relative host IO share allocations.

To avoid extreme behavior from the control algorithm, $w(t)$ is bounded by $[w_{min}, w_{max}]$. The lower bound w_{min} prevents starvation for hosts with very few IO shares. The upper bound w_{max} avoids very long queues at the array, limiting the latency seen by hosts that start issuing requests after a period of inactivity. A reasonable upper bound can be based on typical queue length values in uncontrolled systems, as well as the array configuration and number of hosts.

The latency threshold \mathcal{L} corresponds to the response time that is considered acceptable in the system, and the control algorithm tries to maintain the overall cluster-wide latency close to this value. Testing confirmed our expectation that increasing the array queue length beyond a certain value doesn't lead to increased throughput. Thus, \mathcal{L} can be set to a value which is high enough to ensure that a sufficiently large number of requests can always be pending at the array. We are also exploring automatic techniques for setting this parameter based on long-term observations of latency and throughput. Administrators may alternatively specify \mathcal{L} explicitly, based on cluster-wide requirements, such as supporting latency-sensitive applications, perhaps at the cost of under-utilizing the array in some cases.

Finally, β is set based on the IO shares associated with the host, proportional to the sum of its per-VM shares. It has been shown theoretically in the context of FAST TCP that the equilibrium window size value for different hosts will be proportional to their β parameters [15].

We highlight two properties of the control equation, again relying on formal models and proofs from FAST TCP. First, at equilibrium, the throughput of host i is proportional to β_i/q_i , where β_i is the per-host allocation parameter, and q_i is the queuing delay observed by the host. Second, for a single array with capacity C and latency threshold \mathcal{L} , the window size at equilibrium will be:

$$w_i = \beta_i + \beta_i \frac{C\mathcal{L}}{\sum_j \beta_j} \quad (3)$$

To illustrate the behavior of the control algorithm, we simulated a simple distributed system consisting of a single array and multiple hosts using Yacsim [17]. Each host runs an instance of the algorithm in a distributed manner, and the array services requests with latency based on an exponential distribution with a mean of $1/C$. We conducted a series of experiments with various capacities, workloads, and parameter values.

To test the algorithm's adaptability, we experimented with three hosts using a 1 : 2 : 3 share ratio, $\mathcal{L} = 200$ ms, and an array capacity that changes from 400 req/s to 100 req/s halfway through the experiment. Figure 2 plots the throughput, window size and average latency observed by the hosts for a period of 200 seconds. As expected, the control algorithm drives the system to operate close to the desired

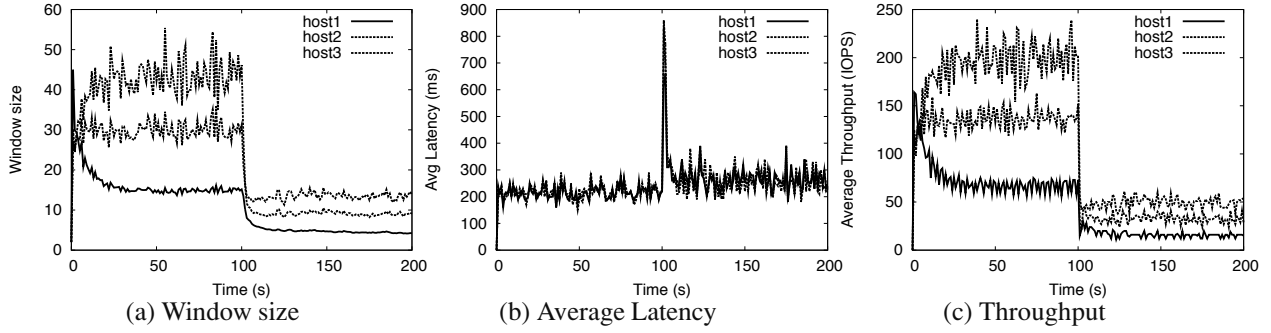


Figure 2: Simulation of three hosts with 1 : 2 : 3 share ratio. Array capacity is reduced from 400 to 100 req/s at $t = 100$ s.

latency threshold \mathcal{L} . We also used the simulator to verify that as \mathcal{L} is varied (100 ms, 200 ms and 300 ms), the system latencies operate close to \mathcal{L} , and that window sizes increase while maintaining their proportional ratio.

3.3 End-to-End Support

PARDA flow control ensures that each host obtains a fair share of storage array capacity proportional to its IO shares. However, our ultimate goal for storage resource management is to provide control over service rates for the applications running in VMs on each host. We use a fair queuing mechanism based on SFQ [10] for our host-level scheduler. SFQ implements proportional-sharing of the host’s issue queue, dividing it among VMs based on their IO shares when there is contention for the host-level queue.

Two key features of the local scheduler are worth noting. First, the scheduler doesn’t strictly partition the host-level queue among VMs based on their shares, allowing them to consume additional slots that are left idle by other VMs which didn’t consume their full allocation. This handles short-term fluctuations in the VM workloads, and provide some statistical multiplexing benefits. Second, the scheduler doesn’t switch between VMs after every IO, instead scheduling a group of IOs per VM as long as they exhibit some spatial locality (within a few MB). These techniques have been shown to improve overall IO performance [3, 13].

Combining a distributed flow control mechanism with a fair local scheduler allows us to provide end-to-end IO allocations to VMs. However, an interesting alternative is to apply PARDA flow control at the VM level, using per-VM latency measurements to control per-VM window sizes directly, independent of how VMs are mapped to hosts. This approach is appealing, but it also introduces new challenges that we are currently investigating. For example, per-VM allocations may be very small, requiring new techniques to support fractional window sizes, as well as efficient distributed methods to compensate for short-term burstiness.

3.4 Handling Bursts

A well-known characteristic of many IO workloads is a bursty arrival pattern—fluctuating resource demand due to device and application characteristics, access locality, and other factors. A high degree of burstiness makes it difficult to provide low latency and achieve proportionate allocation.

In our environment, bursty arrivals generally occur at two distinct time scales: systematic long-term ON-OFF behavior of VMs, and sudden short-term spikes in IO workloads. To handle long-term bursts, we modify the β value for a host based on the utilization of queue slots by its resident VMs. Recall that the host-level parameter β is proportional to the sum of shares of all VMs (if s_i are the shares assigned to VM i , then for host h , $\beta_h = K \times \sum_i s_i$, where K is a normalization constant).

To adjust β , we measure the average number of outstanding IOs per VM, n_k , and each VM’s share of its host window size as w_k , expressed as:

$$w_k = \frac{s_k}{\sum_i s_i} w(t) \quad (4)$$

If $(n_k < w_k)$, we scale the shares of the VM to be $s'_i = n_k \times s_k / w_k$ and use this to calculate β for the host. Thus if a VM is not fully utilizing its window size, we reduce the β value of its host, so other VMs on the same host do not benefit disproportionately due to the under-utilized shares of a colocated idle VM. In general, when one or more VMs become idle, the control mechanism will allow all hosts (and thus all VMs) to proportionally increase their window sizes and exploit the spare capacity.

For short-term fluctuations, we use a burst-aware local scheduler. This scheduler allows VMs to accumulate a bounded number of credits while idle, and then schedule requests in bursts once the VM becomes active. This also improves overall IO efficiency, since requests from a single VM typically exhibit some locality. A number of schedulers support bursty allocations [6, 13, 22]. Our implementation uses SFQ as the local scheduler, but allows a bounded number of IOs to be batched from each VM instead of switching among VMs purely based on their SFQ request tags.

4 Storage-Specific Challenges

Storage devices are stateful and their throughput can be quite variable, making it challenging to apply the latency-based flow control approaches used in networks. Equilibrium may not be reached if different hosts observe very different latencies during overload. Next we discuss three key issues to highlight the differences between storage and network service times.

Request Location. It is well known that the latency of a request can vary from a fraction of a millisecond to tens of milliseconds, based on its location compared to previous requests, as well as caching policies at the array. Variability in seek and rotational delays can cause an order of magnitude difference in service times. This makes it difficult to estimate the baseline IO latency corresponding to the latency with no queuing delay. Thus a sudden change in average latency or in the ratio of current values to the previous average may or may not be a signal for overload. Instead, we look at average latency values in comparison to a latency threshold \mathcal{L} to predict congestion. The assumption is that latencies observed during congestion will have a large *queuing delay* component, outweighing increases due to workload changes (e.g., sequential to random).

Request Type. Write IOs are often returned to the host once the block is written in the controller's NVRAM. Later, they are flushed to disk during the destage process. However, read IOs may need to go to disk more often. Similarly, two requests from a single stream may have widely varying latencies if one hits in the cache and the other misses. In certain RAID systems [5], writes may take four times longer than reads due to parity reads and updates. In general, IOs from a single stream may have widely-varying response times, affecting the latency estimate. Fortunately, a moving average over a sufficiently long period can absorb such variations and provide a more consistent estimate.

IO Size. Typical storage IO sizes range from 512 bytes to 256 KB, or even 1 MB for more recent devices. The estimator needs to be aware of changing IO size in the workload. This can be done by computing latency per 8 KB instead of latency per IO using a linear model with certain fixed costs. Size variance is less of an issue in networks since most packets are broken into MTU-size chunks (typically 1500 bytes) before transmission.

All of these issues essentially boil down to the problem of estimating highly-variable latency and using it as an indicator of array overload. We may need to distinguish between latency changes caused by workload versus those due to the overload at the array. Some of the variation in IO latency can be absorbed by long-term averaging, and by considering latency per fixed IO size instead of per IO request. Also, a sufficiently high baseline latency (the desired oper-

ating point for the control algorithm, \mathcal{L}) will be insensitive to workload-based variations in under-utilized cases.

4.1 Distributed Implementation Issues

We initially implemented PARDA in a completely distributed manner, where each host monitored only its own IO latency to calculate $L(t)$ for Equation 2 (referred to as local latency estimation). However, despite the use of averaging, we found that latencies observed at different hosts were dependent on block-level placement.

We experimented with four hosts, each running one Windows Server 2003 VM configured with a 16 GB data disk created as a contiguous file on the shared LUN. Each VM also has a separate 4 GB system disk. The storage array was an EMC CLARiON CX3-40 (same hardware setup as in Section 5). Each VM executed a 16 KB random read IO workload. Running without any control algorithm, we noticed that the hosts observed average latencies of 40.0, 34.5, 35.0 and 39.5 ms, respectively. Similarly, the throughput observed by the hosts were 780, 910, 920 and 800 IOPS respectively. Notice that hosts two and three achieved better IOPS and lower latency, even though all hosts were issuing exactly the same IO pattern.

We verified that this discrepancy is explained by placement: the VM disks (files) were created and placed in order on the underlying device/LUN, and the middle two virtual disks exhibited better performance compared to the two outer disks. We then ran the control algorithm with latency threshold $\mathcal{L} = 30$ ms and equal β for all hosts. Figure 3 plots the computed window size, latency and throughput over a period of time. The discrepancy in latencies observed across hosts leads to divergence in the system. When hosts two and three observe latencies smaller than \mathcal{L} , they increase their window size, whereas the other two hosts still see latencies higher than \mathcal{L} , causing further window size decreases. This undesirable positive feedback loop leads to a persistent performance gap.

To validate that this effect is due to block placement of VM disks and array level scheduling, we repeated the same experiment using a single 60 GB shared disk. This disk file was opened by all VMs using a "multi-writer" mode. Without any control, all hosts observed a throughput of ~ 790 IOPS and latency of 39 ms. Next we ran with PARDA on the shared disk, again using equal β and $\mathcal{L} = 30$ ms. Figure 4 shows that the window sizes of all hosts are reduced, and the cluster-wide latency stays close to 30 ms.

This led us to conclude that, at least for some disk subsystems, latency observations obtained individually at each host for its IOs are a fragile metric that can lead to divergences. To avoid this problem, we instead implemented a robust mechanism that generates a consistent signal for contention in the entire cluster, as discussed in the next section.

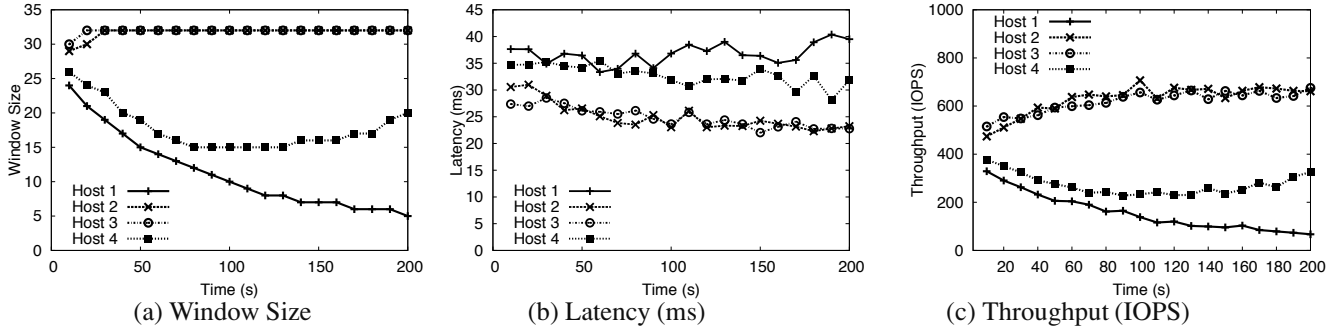


Figure 3: Local $L(t)$ Estimation. Separate VM disks cause window size divergence due to block placement and unfair array scheduling.

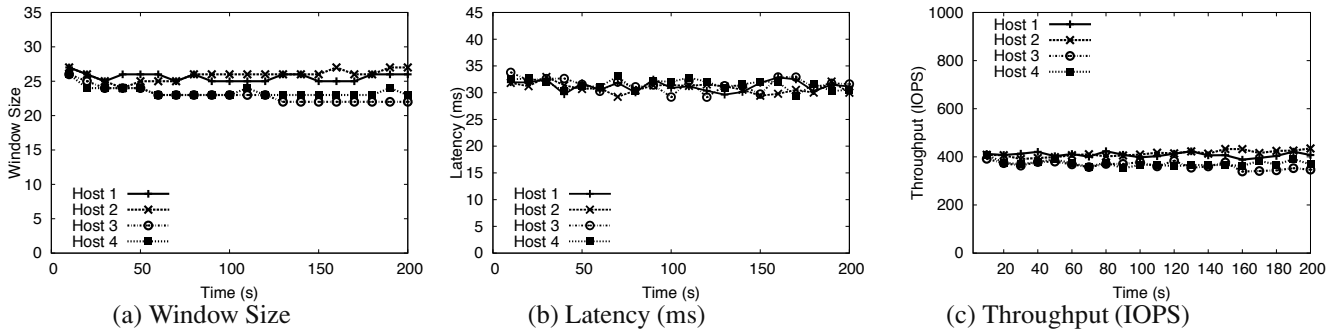


Figure 4: Local $L(t)$ Estimation. VMs use same shared disk, stabilizing window sizes and providing more uniform throughput and latency.

4.2 Latency Aggregation

After experimenting with completely decentralized approaches and encountering the divergence problem detailed above, we implemented a more centralized technique to compute cluster-wide latency as a consistent signal. The aggregation doesn't need to be very accurate, but it should be reasonably consistent across hosts. There are many ways to perform this aggregation, including approximations based on statistical sampling. We discuss two different techniques that we implemented for our prototype.

Network-Based Aggregation. Each host uses a UDP socket to listen for statistics advertised by other hosts. The statistics include the average latency and number of IOs per LUN. Each host either broadcasts its data on a common subnet, or sends it to every other host individually. This is an instance of the general average- and sum-aggregation problem for which multicast-based solutions also exist [29].

Filesystem-Based Aggregation. Since we are trying to control access to a shared filesystem volume (LUN), it is convenient to use the same medium to share the latency statistics among the hosts. We implement a shared file per volume, which can be accessed by multiple hosts simultaneously. Each host owns a single block in the file and periodically writes its average latency and number of IOs for the LUN into that block. Each host reads that file periodically using a single large IO and locally computes the cluster-wide average to use for window size estimation.

In our experiments, we have not observed extremely high variance across per-host latency values, although this seems possible if some workloads are served primarily from the storage array's cache. In any case, we do not anticipate that this would affect PARDA stability or convergence.

5 Experimental Evaluation

In this section, we present the results from a detailed evaluation of PARDA in a real system consisting of up to eight hosts accessing a shared storage array. Each host is a Dell Poweredge 2950 server with 2 Intel Xeon 3.0 GHz dual-core processors, 8 GB of RAM and two Qlogic HBAs connected to an EMC CLARiiON CX3-40 storage array over a Fibre Channel SAN. The storage volume is hosted on a 10-disk RAID-5 disk group on the array.

Each host runs the VMware ESX Server hypervisor [24] with a local instance of the distributed flow control algorithm. The aggregation of average latency uses the filesystem-based implementation described in Section 4.2, with a two-second update period. All PARDA experiments used the smoothing parameters $\alpha = 0.002$ and $\gamma = 0.8$.

Our evaluation consists of experiments that examine five key questions: (1) How does average latency vary with changes in workload? (2) How does average latency vary with load at the array? (3) Can the PARDA algorithm adjust issue queue lengths based on per-host latencies to provide differentiated service? (4) How well can this mechanism

handle bursts and idle hosts? (5) Can we provide end-to-end IO differentiation using distributed flow control together with a local scheduler at each host?

Our first two experiments determine whether average latency can be used as a reliable indicator to detect overload at the storage array, in the presence of widely-varying workloads. The third explores how effectively our control module can adjust host queue lengths to provide coarse-grained fairness. The remaining experiments examine how well PARDA can deal with realistic scenarios that include workload fluctuations and idling, to provide end-to-end fairness to VMs. Throughout this section, we will provide data using a variety of parameter settings to illustrate the adaptability and robustness of our algorithm.

5.1 Latency vs. Workload

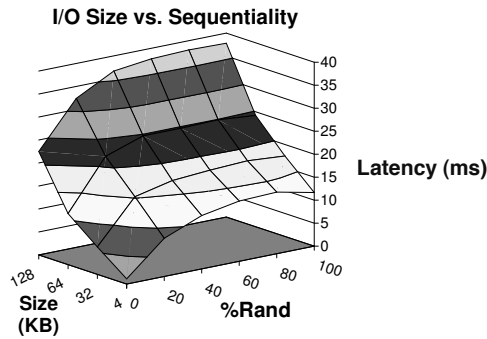


Figure 5: Latency as a function of IO size and sequentiality.

We first consider a single host running one VM executing different workloads, to examine the variation in average latency measured at the host. A Windows Server 2003 VM running *Iometer* [1] is used to generate each workload, configured to keep 8 IOs pending at all times.

We varied three workload parameters: reads – 0 to 100%, IO size – 4, 32, 64, and 128 KB, and sequentiality – 0 to 100%. For each combination, we measured throughput, bandwidth, and the average, min and max latencies.

Over all settings, the minimum latency was observed for the workload consisting of 100% sequential 4 KB reads, while the maximum occurred for 100% random 128 KB writes. Bandwidth varied from 8 MB/s to 177 MB/s. These results show that bandwidth and latency can vary by more than a factor of 20 due solely to workload variation.

Figure 5 plots the average latency (in ms) measured for a VM while varying IO size and degree of sequentiality. Due to space limitations, plots for other parameters have been omitted; additional results and details are available in [11].

There are two main observations: (1) the absolute latency value is not very high for any configuration, and (2) latency usually increases with IO size, but the slope is small because transfer time is usually dominated by seek and ro-

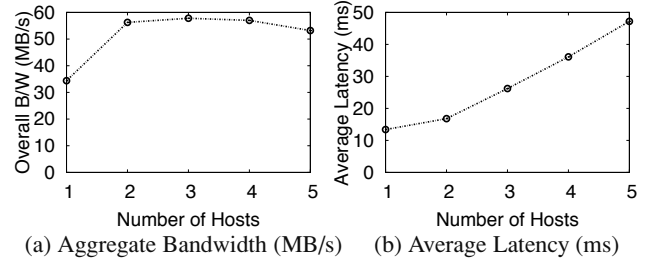


Figure 6: Overall bandwidth and latency observed by multiple hosts as the number of hosts is increased from 1 to 5.

Workload			Phase1			Phase2		
Size	Read	Random	Q	T	L	Q	T	L
16K	70%	60%	32	1160	26	16	640	24
16K	100%	100%	32	880	35	32	1190	27
8K	75%	0%	32	1280	25	16	890	17
8K	90%	100%	32	900	36	32	1240	26

Table 2: Throughput (T IOPS) and latencies (L ms) observed by four hosts for different workloads and queue lengths (Q).

tational delays. This suggests that array overload can be detected by using a fairly high latency threshold value.

5.2 Latency vs. Queue Length

Next we examine how IO latency varies with increases in overall load (queue length) at the array. We experimented with one to five hosts accessing the same array. Each host generates a uniform workload of 16 KB IOs, 67% reads and 70% random, keeping 32 IOs outstanding. Figure 6 shows the aggregate throughput and average latency observed in the system, with increasing contention at the array. Throughput peaks at three hosts, but overall latency continues to increase with load. Ideally, we would like to operate at the lowest latency where bandwidth is high, in order to fully utilize the array without excessive queuing delay.

For uniform workloads, we also expect a good correlation between queue size and overall throughput. To verify this, we configured seven hosts to access a 400 GB volume on a 5-disk RAID-5 disk group. Each host runs one VM with an 8 GB virtual disk. We report data for a workload of 32 KB IOs with 67% reads, 70% random and 32 IOs pending. Figure 7 presents results for two different static host-level window size settings: (a) 32 for all hosts and (b) 16 for hosts 5, 6 and 7.

We observe that the VMs on the throttled hosts receive approximately half the throughput (~ 42 IOPS) compared to other hosts (~ 85 IOPS) and their latency (~ 780 ms) is doubled compared to others (~ 360 ms). Their reduced performance is a direct result of throttling, and the increased latency arises from the fact that a VM’s IOs were queued at its host. The device latency measured at the hosts (as opposed to in the VM, which would include time spent in host queues) is similar for all hosts in both experiments. The

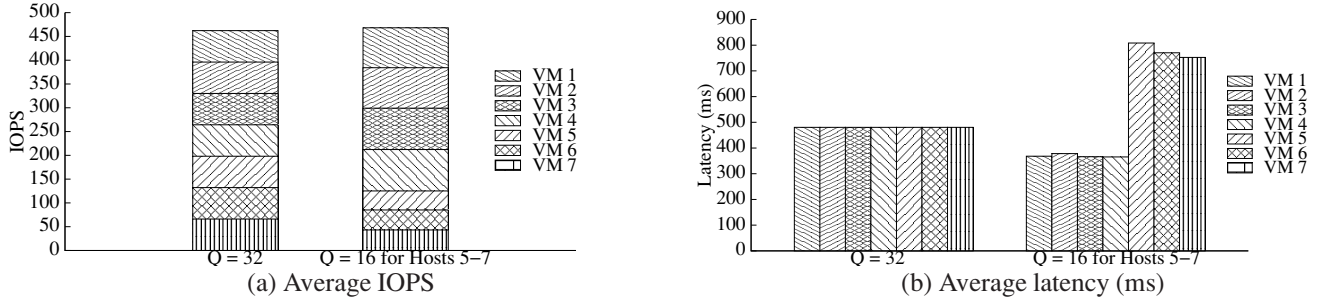


Figure 7: VM bandwidth and latency observed when queue length $Q = 32$ for all hosts, and when $Q = 16$ for some hosts.

overall latency decreases when one or more hosts are throttled, since there is less load on the array. For example, in the second experiment, the overall average latency changes from ~ 470 ms at each host to ~ 375 ms at each host when the window size is 16 for hosts 5, 6, and 7.

We also experimented with four hosts sending different workloads to the array while we varied their queue lengths in two phases. Table 2 reports the workload description and corresponding throughput and latency values observed at the hosts. In phase 1, each host has a queue length of 32 while in phase 2, we lowered the queue length for two of the hosts to 16. This experiment demonstrates two important properties. First, overall throughput reduces roughly in proportion to queue length. Second, if a host is receiving higher throughput at some queue length Q due to its workload being treated preferentially, then even for a smaller queue length $Q/2$, the host still obtains preferential treatment from the array. This is desirable because overall efficiency is improved by giving higher throughput to request streams that are less expensive for the array to process.

5.3 PARDA Control Method

In this section, we evaluate PARDA by examining fairness, latency threshold effects, robustness with non-uniform workloads, and adaptation to capacity changes.

5.3.1 Fairness

We experimented with identical workloads accessing 16 GB virtual disks from four hosts with equal β values. This is similar to the setup that led to divergent behavior in Figure 3. Using our filesystem-based aggregation, PARDA converges as desired, even in the presence of different latency values observed by hosts. Table 3 presents results for this workload without any control, and with PARDA using equal shares for each host; plots are omitted due to space constraints. With PARDA, latencies drop, making the overall average close to the target \mathcal{L} . The aggregate throughput achieved by all hosts is similar with and without PARDA, exhibiting good work-conserving behavior. This demonstrates that the algorithm works correctly in the simple case of equal shares and uniform workloads.

Host	Uncontrolled		β	PARDA $\mathcal{L} = 30$ ms	
	IOPS	Latency (ms)		IOPS	Latency (ms)
1	780	41	1	730	34
2	900	34	1	890	29
3	890	35	1	930	29
4	790	40	1	800	33
Aggregate	3360	Avg = 37		3350	Avg = 31

Table 3: Fairness with 16 KB random reads from four hosts.

Next, we experimented with a share ratio of 1 : 1 : 2 : 2 for four hosts, setting $\mathcal{L} = 25$ ms, shown in Figure 8. PARDA converges on window sizes for hosts 1 and 2 that are roughly half those for hosts 3 and 4, demonstrating good fairness. The algorithm also successfully converges latencies to \mathcal{L} . Finally, the per-host throughput levels achieved while running this uniform workload also roughly match the specified share ratio. The remaining differences are due to some hosts obtaining better throughput from the array, even with the same window size. This reflects the true IO costs as seen by the array scheduler; since PARDA operates on window sizes, it maintains high efficiency at the array.

5.3.2 Effect of Latency Threshold

Recall that \mathcal{L} is the desired latency value at which the array provides high throughput but small queuing delay. Since PARDA tries to operate close to \mathcal{L} , an administrator can control the overall latencies in a cluster, bounding IO times for latency-sensitive workloads such as OLTP. We investigated the effect of the threshold setting by running PARDA with different \mathcal{L} values. Six hosts access the array concurrently, each running a VM with a 16 GB disk performing 16 KB random reads with 32 outstanding IOs.

Host	IOPS	Latency (ms)	Host	IOPS	Latency (ms)
1	525	59	4	560	57
2	570	55	5	430	77
3	570	55	6	500	62

Table 4: Uncontrolled 16 KB random reads from six hosts.

We first examine the throughput and latency observed in the uncontrolled case, presented in Table 4. In Figure 9, we enable the control algorithm with $\mathcal{L} = 30$ ms and equal shares, stopping one VM each at times $t = 145$ s, $t = 220$ s

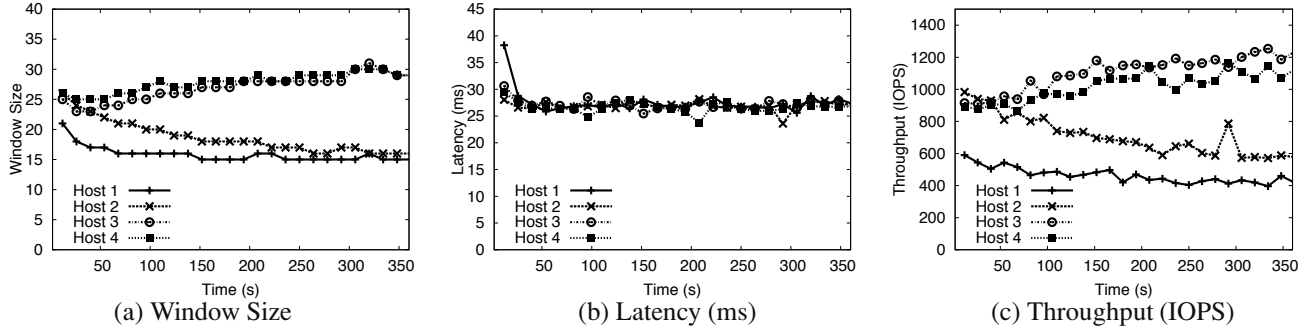


Figure 8: PARDA Fairness. Four hosts each run a 16 KB random read workload with β values of 1 : 1 : 2 : 2. Window sizes allocated by PARDA are in proportion to β values, and latency is close to the specified threshold $\mathcal{L} = 25$ ms.

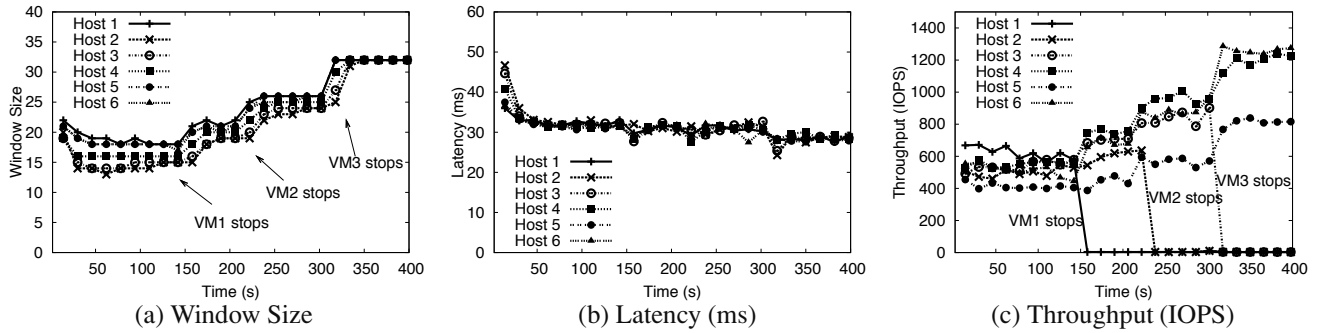


Figure 9: PARDA Adaptation. Six hosts each run a 16 KB random read workload, with equal β values and $\mathcal{L} = 30$ ms. VMs are stopped at $t = 145$ s, $t = 220$ s and $t = 310$ s, and window sizes adapt to reflect available capacity.

and $t = 310$ s. Comparing the results we can see the effect of the control algorithm on performance. Without PARDA, the system achieves a throughput of 3130 IOPS at an average latency of 60 ms. With $\mathcal{L} = 30$ ms, the system achieves a throughput of 3150 IOPS, while operating close to the latency threshold. Other experiments with different threshold values, such as those shown in Figure 10 ($\mathcal{L} = 40$ ms) and Figure 12 ($\mathcal{L} = 25$ ms), confirm that PARDA is effective at maintaining latencies near \mathcal{L} .

These results demonstrate that PARDA is able to control latencies by throttling IO from hosts. Note the different window sizes at which hosts operate for different values of \mathcal{L} . Figure 9(a) also highlights the adaptation of window sizes, as more capacity becomes available at the array when VMs are turned off at various points in the experiment. The ability to detect capacity changes through changes in latency is an important dynamic property of the system.

5.3.3 Non-Uniform Workloads

To test PARDA and its robustness with mixed workloads, we ran very different workload patterns at the same time from our six hosts. Table 5 presents the uncontrolled case.

Next, we enable PARDA with $\mathcal{L} = 40$ ms, and assign shares in a 2 : 1 : 2 : 1 : 2 : 1 ratio for hosts 1 through 6 respectively, plotted in Figure 10. Window sizes are differentiated between hosts with different shares. Hosts with more

Host	Size	Read	Random	IOPS	Latency (ms)
1	4K	100%	100%	610	51
2	8K	50%	0%	660	48
3	8K	100%	100%	630	50
4	8K	67%	60%	670	47
5	16K	100%	100%	490	65
6	16K	75%	70%	520	60

Table 5: Uncontrolled access by mixed workloads from six hosts.

shares reach a window size of 32 (the upper bound, w_{max}) and remain there. Other hosts have window sizes close to 19. The average latency observed by the hosts remains close to \mathcal{L} , as shown in Figure 10(b). The throughput observed by hosts follows roughly the same pattern as window sizes, but is not always proportional because of array scheduling and block placement issues. We saw similar adaptation in window sizes and latency when we repeated this experiment using $\mathcal{L} = 30$ ms (plots omitted due to space constraints).

5.3.4 Capacity Changes

Storage capacity can change dramatically due to workload changes or array accesses by uncontrolled hosts external to PARDA. We have already demonstrated in Section 5.3.2 that our approach is able to absorb any spare capacity that becomes available. To test the ability of the control algorithm to handle decreases in capacity, we conducted an experiment starting with the first five hosts from the previous

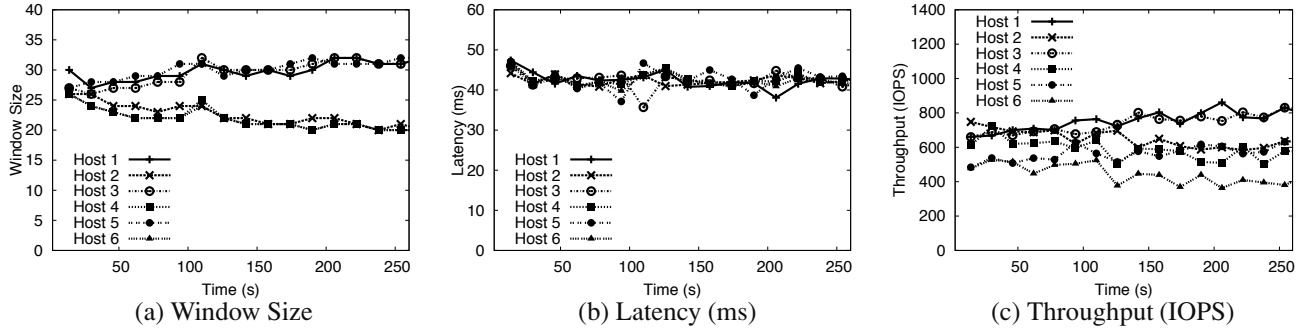


Figure 10: Non-Uniform Workloads. PARDA control with $\mathcal{L} = 40$ ms. Six hosts run mixed workloads, with β values 2 : 1 : 2 : 1 : 2 : 1.

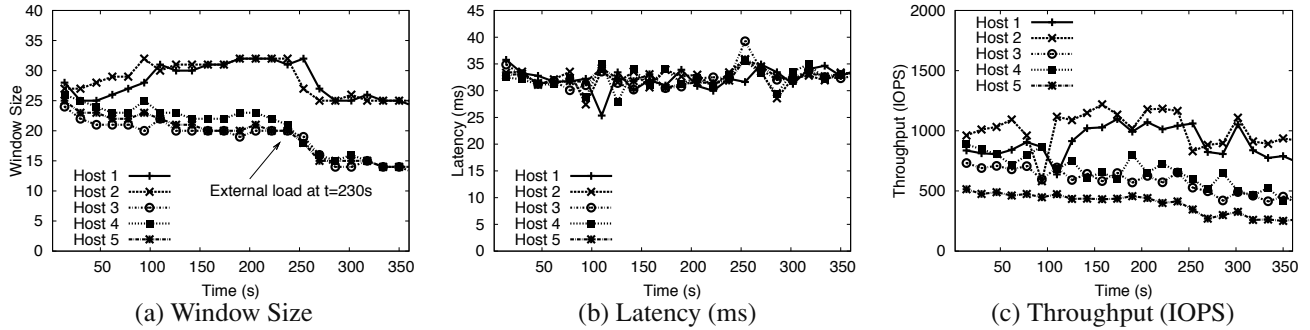


Figure 11: Capacity Fluctuation. Uncontrolled external host added at $t = 230$ s. PARDA-controlled hosts converge to new window sizes.

experiment. At time $t = 230$ s, we introduce a sixth host that is not under PARDA control. This uncontrolled host runs a Windows Server 2003 VM issuing 16 KB random reads to a 16 GB virtual disk located on the same LUN as the others.

With $\mathcal{L} = 30$ ms and a share ratio of 2 : 2 : 1 : 1 : 1 for the PARDA-managed hosts, Figure 11 plots the usual metrics over time. At $t = 230$ s, the uncontrolled external host starts, thereby reducing available capacity for the five controlled hosts. The results indicate that as capacity changes, the hosts under control adjust their window sizes in proportion to their shares, and observe latencies close to \mathcal{L} .

5.4 End-to-End Control

We now present an end-to-end test where multiple VMs run a mix of realistic workloads with different shares. We use Filebench [20], a well-known IO modeling tool, to generate an OLTP workload similar to TPC-C. We employ four VMs running Filebench, and two generating 16 KB random reads. A pair of Filebench VMs are placed on each of two hosts, whereas the micro-benchmark VMs occupy one host each. This is exactly the same experiment discussed in Section 2; data for the uncontrolled baseline case is presented in Table 1. Recall that without PARDA, hosts 1 and 2 obtain similar throughput even though the overall sum of their VM shares is different. Table 6 provides setup details and reports data using PARDA control. Results for the OLTP VMs are presented as Filebench operations per second (Ops/s).

Host	VM Type	s_1, s_2	β_h	VM1	VM2	T_h
1	2×OLTP	20, 10	6	1266 Ops/s	591 Ops/s	1857
2	2×OLTP	10, 10	4	681 Ops/s	673 Ops/s	1316
3	1×Micro	20	4	740 IOPS	n/a	740
4	1×Micro	10	2	400 IOPS	n/a	400

Table 6: PARDA end-to-end control for Filebench OLTP and micro-benchmark VMs issuing 16 KB random reads. Configured shares (s_i), host weights (β_h), Ops/s for Filebench VMs and IOPS (T_h for hosts) are respected across hosts. $\mathcal{L} = 25$ ms, $w_{max} = 64$.

We run PARDA ($\mathcal{L} = 25$ ms) with host weights (β_h) set according to shares of their VMs ($\beta_h = 6 : 4 : 4 : 2$ for hosts 1 to 4). The maximum window size w_{max} is 64 for all hosts. The OLTP VMs on host 1 receive 1266 and 591 Ops/s, matching their 2 : 1 share ratio. Similarly, OLTP VMs on host 2 obtain 681 and 673 Ops/s, close to their 1 : 1 share ratio. Note that the overall Ops/s for hosts 1 and 2 have a 3 : 2 ratio, which is not possible in an uncontrolled scenario. Figure 12 plots the window size, latency and throughput observed by the hosts. We note two key properties: (1) window sizes are in proportion to the overall β values and (2) each VM receives throughput in proportion to its shares. This shows that PARDA provides the strong property of enforcing VM shares, independent of their placement on hosts. The local SFQ scheduler divides host-level capacity across VMs in a fair manner, and together with PARDA, is able to provide effective end-to-end isolation among VMs. We also modified one VM workload during the experiment

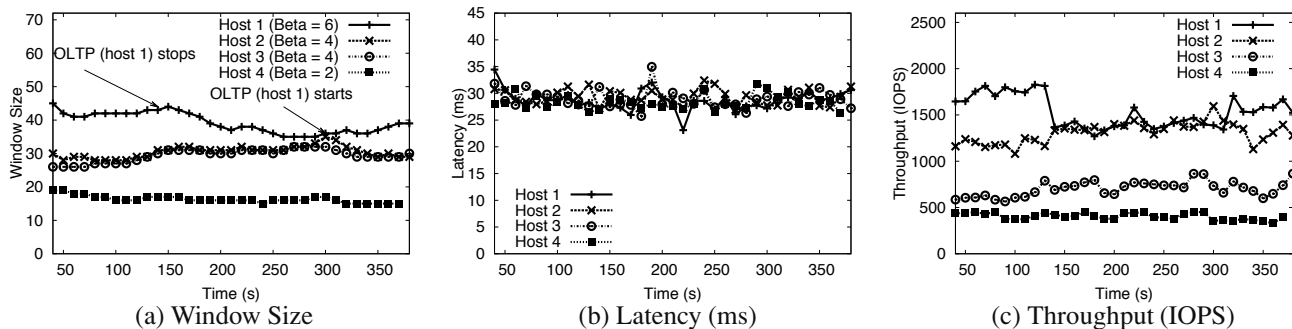


Figure 12: PARDA End-to-End Control. VM IOPS are proportional to shares. Host window sizes are proportional to overall β values.

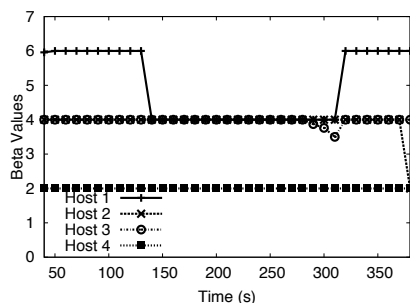


Figure 13: Handling Bursts. One OLTP workload on host 1 stops at $t = 140$ s and restarts at $t = 310$ s. The β of host 1 is adjusted and window sizes are recomputed using the new β value.

to test our burst-handling mechanism, which we discuss in the next section.

5.5 Handling Bursts

Earlier we showed that PARDA maintains high utilization of the array even when some hosts idle, by allowing other hosts to increase their window sizes. However, if one or more VMs become idle, the overall β of the host must be adjusted, so that backlogged VMs on the same host don't obtain an unfair share of the current capacity. Our implementation employs the technique described in Section 3.4.

We experimented with dynamically idling one of the OLTP VM workloads running on host 1 from the previous experiment presented in Figure 12. The VM workload is stopped at $t = 140$ s and resumed at $t = 310$ s. Figure 13 shows that the β value for host 1 adapts quickly to the change in the VM workload. Figure 12(a) shows that the window size begins to decrease according to the modified lower value of $\beta = 4$ starting from $t = 140$ s. By $t = 300$ s, window sizes have converged to a 1 : 2 ratio, in line with aggregate host shares. As the OLTP workload becomes active again, the dynamic increase in the β of host 1 causes its window size to grow. This demonstrates that PARDA ensures fairness even in the presence of non-backlogged workloads, a highly-desirable property for shared storage access.

Host	VM Type	Uncontrolled			PARDA		
		OPM	Avg Lat	T_h, L_h	β_h	OPM	Avg Lat
1	SQL1	8799	213	615, 20.4	1	6952	273
2	SQL2	8484	221	588, 20.5	4	12356	151

Table 7: Two SQL Server VMs with 1 : 4 share ratio, running with and without PARDA. Host weights (β_h) and OPM (orders/min), IOPS (T_h for hosts) and latencies (Avg Lat for database operations, L_h for hosts, in ms). $\mathcal{L} = 15$ ms, $w_{max} = 32$.

5.6 Enterprise Workloads

To test PARDA with more realistic enterprise workloads, we experimented with two Windows Server 2003 VMs, each running a Microsoft SQL Server 2005 Enterprise Edition database. Each VM is configured with 4 virtual CPUs, 6.4 GB of RAM, a 10 GB system disk, a 250 GB database disk, and a 50 GB log disk. The database virtual disks are hosted on an 800 GB RAID-0 LUN with 6 disks; log virtual disks are placed on a 100 GB RAID-0 LUN with 10 disks. We used the Dell DVD store (DS2) database test suite, which implements a complete online e-commerce application, to stress the SQL databases [7]. We configured a 15 ms latency threshold, and ran one VM per host, assigning shares in a 1 : 4 ratio.

Table 7 reports the parameters and the overall application performance for the two SQL Server VMs. Without PARDA, both VMs have similar performance in terms of both orders per minute (OPM) and average latency. When running with PARDA, the VM with higher shares obtains roughly twice the OPM throughput and half the average latency. The ratio isn't 1 : 4 because the workloads are not always backlogged, and the VM with higher shares can't keep its window completely full.

Figure 14 plots the window size, latency and throughput observed by the hosts. As the overall latency decreases, PARDA is able to assign high window sizes to both hosts. When latency increases, the window sizes converge to be approximately proportional to the β values. Figure 15 shows the β values for the hosts while the workload is running, and highlights the fact that the SQL Server VM on host 2 cannot always maintain enough pending IOs to fill

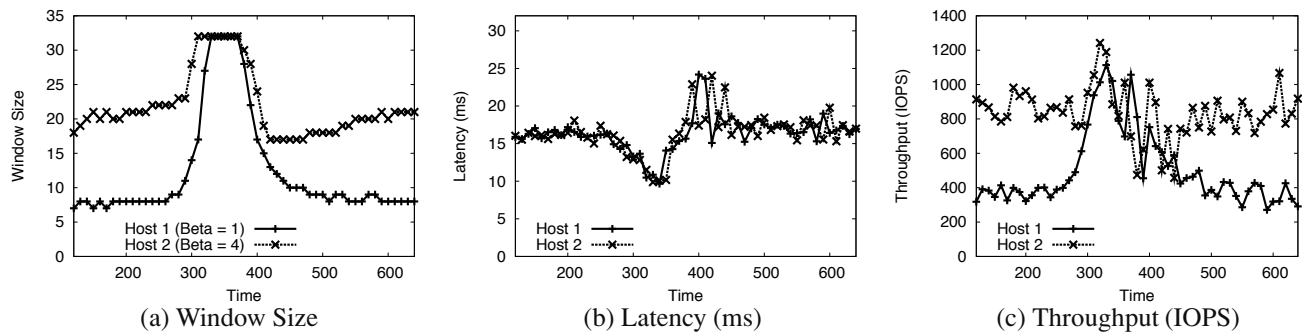


Figure 14: Enterprise Workload. Host window sizes and IOPS for SQL Server VMs are proportional to their overall β values whenever the array resources are contended. Between $t = 300$ s and $t = 380$ s, hosts get larger window sizes since the array is not contended.

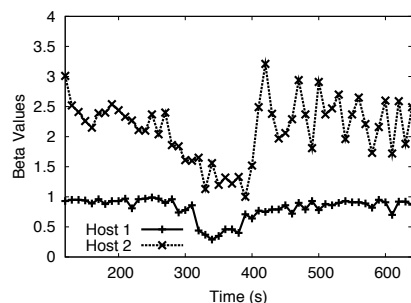


Figure 15: Dynamic β Adjustment. β values for hosts running SQL Server VMs fluctuate as pending IO counts change.

its window. This causes the other VM on host 1 to pick up the slack and benefit from increased IO throughput.

6 Related Work

The research literature contains a large body of work related to providing quality of service in both networks and storage systems, stretching over several decades. Numerous algorithms for network QoS have been proposed, including many variants of fair queuing [2, 8, 10]. However, these approaches are suitable only in centralized settings where a single controller manages all requests for resources. Stoica proposed QoS mechanisms based on a stateless core [23], where only edge routers need to maintain per-flow state, but some minimal support is still required from core routers.

In the absence of such mechanisms, TCP has been serving us quite well for both flow control and congestion avoidance. Commonly-deployed TCP variants use per-flow information such as estimated round trip time and packet loss at each host to adapt per-flow window sizes to network conditions. Other proposed variants [9] require support from routers to provide congestion signals, inhibiting adoption.

FAST-TCP [15] provides a purely latency-based approach to improving TCP's throughput in high bandwidth-delay product networks. In this paper we adapt some of

the techniques used by TCP and its variants to perform flow control in distributed storage systems. In so doing, we have addressed some of the challenges that make it non-trivial to employ TCP-like solutions for managing storage IO.

Many storage QoS schemes have also been proposed to provide differentiated service to workloads accessing a single disk or storage array [4, 13, 14, 16, 25, 30]. Unfortunately, these techniques are centralized, and generally require full control over all IO. Proportionate bandwidth allocation algorithms have also been developed for distributed storage systems [12, 26]. However, these mechanisms were designed for brick-based storage, and require each storage device to run an instance of the scheduling algorithm.

Deployments of virtualized systems typically have no control over storage array firmware, and don't use a central IO proxy. Most commercial storage arrays offer only limited, proprietary quality-of-service controls, and are treated as black boxes by the virtualization layer. Triage [18] is one control-theoretic approach that has been proposed for managing such systems. Triage periodically observes the utilization of the system and throttles hosts using bandwidth caps to achieve a specified share of available capacity. This technique may underutilize array resources, and relies on a central controller to gather statistics, compute an on-line system model, and re-assign bandwidth caps to hosts. Host-level changes must be communicated to the controller to handle bursty workloads. In contrast, PARDA only requires very light-weight aggregation and per-host measurement and control to provide fairness with high utilization.

Friendly VMs [31] propose cooperative fair sharing of CPU and memory in virtualized systems leveraging feedback-control models. Without relying on a centralized controller, each "friendly" VM adapts its own resource consumption based on congestion signals, such as the relative progress of its virtual time compared to elapsed real time, using TCP-like AIMD adaptation. PARDA applies similar ideas to distributed storage resource management.

7 Conclusions

In this paper, we studied the problem of providing coarse-grained fairness to multiple hosts sharing a single storage system in a distributed manner. We propose a novel software system, PARDA, which uses average latency as an indicator for array overload and adjusts per-host issue queue lengths in a decentralized manner using flow control.

Our evaluation of PARDA in a hypervisor shows that it is able to provide fair access to the array queue, control overall latency close to a threshold parameter and provide high throughput in most cases. Moreover, combined with a local scheduler, PARDA is able to provide end-to-end prioritization of VM IOs, even in presence of variable workloads.

As future work, we are trying to integrate soft limits and reservations to provide a complete IO management framework. We would also like to investigate applications of PARDA to other non-storage systems where resource management must be implemented in a distributed fashion.

Acknowledgements

Thanks to Tim Mann, Minwen Ji, Anne Holler, Neeraj Goyal, Narasimha Raghunandana and our shepherd Jiri Schindler for valuable discussions and feedback. Thanks also to Chethan Kumar for help with experimental setup.

References

- [1] Iometer. <http://www.iometer.org>.
- [2] BENNETT, J. C. R., AND ZHANG, H. WF^2Q : Worst-case fair weighted fair queueing. In *Proc. of IEEE INFOCOM '96* (March 1996), pp. 120–128.
- [3] BRUNO, J., BRUSTOLONI, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. Disk scheduling with quality of service guarantees. In *Proc. of the IEEE Int'l Conf. on Multimedia Computing and Systems, Volume 2* (1999), IEEE Computer Society.
- [4] CHAMBLISS, D. D., ALVAREZ, G. A., PANDEY, P., JADAV, D., XU, J., MENON, R., AND LEE, T. P. Performance virtualization for large-scale storage systems. In *Symposium on Reliable Distributed Systems* (October 2003), pp. 109–118.
- [5] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys* 26, 2 (1994).
- [6] CRUZ, R. L. Quality of service guarantees in virtual circuit switched networks. *IEEE Journal on Selected Areas in Communications* 13, 6 (1995), 1048–1056.
- [7] DELL, INC. DVD Store. <http://delltechcenter.com/page/DVD+store>.
- [8] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queuing algorithm. *Journal of Internetworking Research and Experience* 1, 1 (September 1990), 3–26.
- [9] FLOYD, S., AND JACOBSON, V. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking* 1, 4 (1993), 397–413.
- [10] GOYAL, P., VIN, H. M., AND CHENG, H. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking* 5, 5 (1997).
- [11] GULATI, A., AND AHMAD, I. Towards distributed storage resource management using flow control. In *Proc. of First International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability* (2008).
- [12] GULATI, A., MERCHANT, A., AND VARMAN, P. dClock: Distributed QoS in heterogeneous resource environments. In *Proc. of ACM PODC (short paper)* (August 2007).
- [13] GULATI, A., MERCHANT, A., AND VARMAN, P. pClock: An arrival curve based approach for QoS in shared storage systems. In *Proc. of ACM SIGMETRICS* (June 2007), pp. 13–24.
- [14] HUANG, L., PENG, G., AND CHIU, T. Multi-dimensional storage virtualization. In *Proc. of ACM SIGMETRICS* (June 2004).
- [15] JIN, C., WEI, D., AND LOW, S. FAST TCP: Motivation, Architecture, Algorithms, Performance. *Proceedings of IEEE INFOCOM '04* (March 2004).
- [16] JIN, W., CHASE, J. S., AND KAUR, J. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS* (June 2004).
- [17] JUMP, J. R. Yacsim reference manual. <http://www.owl.net.rice.edu/~elec428/yacsim/yacsim.man.ps>.
- [18] KARLSSON, M., KARAMANOLIS, C., AND ZHU, X. Triage: Performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage* 1, 4 (2005), 457–480.
- [19] LUMB, C., MERCHANT, A., AND ALVAREZ, G. Façade: Virtual storage devices with performance guarantees. *Proc. of File and Storage Technologies (FAST)* (March 2003).
- [20] MCDUGALL, R. Filebench: A prototype model based workload for file systems, work in progress. http://solarisinternals.com/si/tools/filebench/filebench_nasconf.pdf.
- [21] POVZNER, A., KALDEWEY, T., BRANDT, S., GOLDING, R., WONG, T. M., AND MALTZAHN, C. Efficient guaranteed disk request scheduling with Fahrrad. *SIGOPS Oper. Syst. Rev.* 42, 4 (2008), 13–25.
- [22] SARIOWAN, H., CRUZ, R. L., AND POLYZOS, G. C. Scheduling for quality of service guarantees via service curves. In *Proceedings of the International Conference on Computer Communications and Networks* (1995), pp. 512–520.
- [23] STOICA, I., SHENKER, S., AND ZHANG, H. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high speed networks. *IEEE/ACM Transactions on Networking* 11, 1 (2003), 33–46.
- [24] VMWARE, INC. *Introduction to VMware Infrastructure*. 2007. <http://www.vmware.com/support/pubs/>.
- [25] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: performance insulation for shared storage servers. In *Proc. of File and Storage Technologies (FAST)* (Feb 2007).
- [26] WANG, Y., AND MERCHANT, A. Proportional-share scheduling for distributed storage systems. In *Proc. of File and Storage Technologies (FAST)* (Feb 2007).
- [27] WONG, T. M., GOLDING, R. A., LIN, C., AND BECKER-SZENDY, R. A. Zygaria: Storage performance as a managed resource. In *Proc. of Real-Time and Embedded Technology and Applications Symposium* (April 2006), pp. 125–34.
- [28] WU, J. C., AND BRANDT, S. A. The design and implementation of Aqua: an adaptive quality of service aware object-based storage device. In *Proc. of MSST* (May 2006), pp. 209–18.
- [29] YALAGANDULA, P. A scalable distributed information management system. In *Proc. of SIGCOMM* (2004), pp. 379–390.
- [30] ZHANG, J., SIVASUBRAMANIAM, A., WANG, Q., RISKA, A., AND RIEDEL, E. Storage performance virtualization via throughput and latency control. In *Proc. of MASCOTS* (September 2005).
- [31] ZHANG, Y., BESTAVROS, A., GUIRGUIS, M., MATTA, I., AND WEST, R. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *Proc. of Intl. Conference on Virtual Execution Environments (VEE)* (June 2005).

CA-NFS: A Congestion-Aware Network File System

Alexandros Batsakis
NetApp
Johns Hopkins University

Randal Burns
Johns Hopkins University

Arkady Kanevsky
NetApp

James Lentini
NetApp

Thomas Talpey
NetApp

Abstract

We develop a holistic framework for adaptively scheduling asynchronous requests in distributed file systems. The system is holistic in that it manages all resources, including network bandwidth, server I/O, server CPU, and client and server memory utilization. It accelerates, defers, or cancels asynchronous requests in order to improve application-perceived performance directly. We employ congestion pricing via online auctions to coordinate the use of system resources by the file system clients so that they can detect shortages and adapt their resource usage. We implement our modifications in the Congestion-Aware Network File System (CA-NFS), an extension to the ubiquitous network file system (NFS). Our experimental result shows that CA-NFS results in a 20% improvement in execution times when compared with NFS for a variety of workloads.

1 Introduction

Distributed file system clients consume server and network resources without consideration for how their operations interfere with their future requests and other clients. Each client request incurs a cost to the system, expressed in increased load to one or more of its resources. As more capacity, more workload, or more users are added congestion rises, and all client operations share the cost in delayed execution. However, clients remain oblivious to the congestion level of the system resources.

When the system is under congestion, network file servers try to maximize throughput across clients, assuming that their benefit increases with the flow rate. This practice does not correspond well with application-perceived performance because it fails to distinguish the urgency and relative priority of file system operations across the client population. From the server's perspective, all client operations at any given time are equally important. This is a fallacy. File system opera-

tions come at different priorities implicitly. While some need to be performed on demand, many can be deferred. Synchronous client operations (metadata, reads) benefit more from timely execution than asynchronous operations (most writes, read-aheads), because the former block the calling application until completion. Also, certain asynchronous operations are more urgent than others depending on the client's state. For example, when a client's memory consumption is high, all of its write operations become synchronous, leading to a degradation in system performance.

In this paper, we develop a performance management framework for distributed file systems that dynamically assesses system load, manages system resources, and schedules asynchronous client operations. When the system resources approach critical capacity, we apply priority scheduling, preferring blocking to non-blocking requests, and priority inheritance, *e.g.* performing writes that block reads at high priority, so that non-time-critical (asynchronous) I/O traffic does not interfere with on-demand (synchronous) requests. On the other hand, if the system load is low, we perform asynchronous operations more aggressively in order to avoid the possibility of performing the same operations at a later time, when the server resources will be congested.

The framework is based on a *holistic* congestion pricing mechanism that incorporates all critical resources among all clients and servers, from client caches to server disk subsystems. Holistic goes beyond end-to-end in that it balances resource usage across multiple clients and servers. (End-to-end also connotes network endpoints and holistic management goes from client applications to server disk systems.) The holistic approach allows the system to address different bottlenecks in different configurations and respond to changing resource limitations over time.

Servers encode their resource constraints by increasing or decreasing the price of asynchronous reads and writes in the system in order to "push back" at clients.

As the server prices increase, the clients that are not resource constrained will defer asynchronous operations for a later time and, thus, reduce their presented load. This helps to avoid congestion in the network and server I/O system caused by non-critical operations.

The underlying pricing algorithm, based on resource utilization, provides a $\log-k$ competitive solution to resource pricing when compared with an offline algorithm that “knows” all future requests. In contrast to heuristic methods for moving thresholds, this approach is system and workload independent.

We evaluate our proposed changes in CA-NFS (Congestion-Aware Network File System), an extension of the NFS protocol, implemented as modifications to the Linux NFS client, server, and memory manager. Experimental results show that CA-NFS outperforms NFS and improves application-perceived performance by more than 20% in a wide variety of workloads.

2 System Operation

In this section, we give the intuition behind scheduling asynchronous operations and the effect these have on system resource utilization. We then demonstrate how clients adapt their behavior using pricing and auctions.

2.1 Asynchronous Writes

The effectiveness of asynchronous write operations depends on the client’s current memory state. Writes are asynchronous only if there is available memory; a system that cannot allocate memory to a write, blocks that write until memory can be freed. This hampers performance severely because all subsequent writes become effectively synchronous. It also has an adverse effect on reads. All pending writes that must be written to storage interfere with concurrent reads, which results in queuing delays at the network and disk.

CA-NFS changes the way that asynchronous writes are performed compared to regular NFS. NFS clients write data to the server’s memory immediately upon receiving a `write()` system call and also buffer the write data in local memory. The buffered pages are marked as dirty at both the client and the server. To harden these data to disk, the client sends a `commit` message to the server. The decision of when to commit the data to the server depends on several factors. Traditionally, systems used a periodic update policy in which individual dirty blocks are flushed when their age reaches a predefined limit [32]. Modern systems destage dirty pages when the number of dirty pages in memory exceeds a certain percentage (flushing point), which is typically a small fraction of the available memory (e.g. 10%). Then, a daemon wakes up and starts flushing dirty pages until an adequate number of pages have reached stable storage.

In contrast to regular NFS, CA-NFS clients adapt their asynchronous write behavior by either *deferring* or *accelerating* a write. CA-NFS clients accelerate writes by forcing the CA-NFS server to sync the data to stable storage so that the client does not need to buffer all of the corresponding dirty pages. The idea behind write acceleration is that if the server resource utilization is low, there is no need to defer the commit to a later time. Also, clients may elect to accelerate writes in order to preserve their cache contents and maintain a high cache hit rate. Note that accelerating a write does not make the write operation synchronous. Instead, it invokes the write-back daemon at the client immediately.

Write acceleration possibly increases the server disk utilization and uses network bandwidth immediately. In write-behind systems, many writes are canceled before they reach the server [5, 34], e.g. writing the same file page repeatedly, or creating and deleting a temporary file. Thus, the load imposed to the server as a result of write acceleration could be avoided. However, write acceleration has almost no negative effect on system performance, because CA-NFS accelerates writes only when the server load is low.

Deferring a write avoids copying dirty data to server memory upon receiving a write request. Instead, clients keep data in local memory only, until the price of using the server resources is low. Clients price asynchronous writes based on their ability to cache writes, i.e. available memory. A client with scarce memory, because of write deferral, will increase its local price for writes so that its buffered pages will be transferred to the server as soon as possible. To make write deferral possible, we modify the operation of the write-back daemon on the clients by dynamically changing the flushing point value based on the pricing mechanism to dictate when the write-back of dirty pages should begin.

Deferring a write consumes client memory with dirty pages, saves server memory, and delays the consumption of network bandwidth and server disk I/O. However, it faces the risk of imposing higher latency for subsequent synchronous `commit` operations. This is because a file sync may require a network transfer of the dirty buffers from the client to server memory. Note that deferring a write does not guarantee that the server price for the same operation will be lower in the future. Instead, this policy gives priority to operations originating from resource-constrained clients.

CA-NFS follows NFS’s close-to-open consistency model. Deferring or accelerating writes does not violate the consistency semantics of NFS, because CA-NFS does not change the semantics of the `COMMIT` operation. Asynchronous write-back in NFS includes a deadline that, when it elapses, escalates the operation to a synchronous write. CA-NFS does the same.

The server prices asynchronous writes based on its memory, disk and network utilization. If the server memory contains blocks that are currently accessed by clients, setting high prices forces clients to defer writes in order to preserve cache contents and maintain a high cache hit rate. Also, if the disk or network resources are heavily utilized, CA-NFS defers writes until the load decreases, to avoid queuing delays because of pending writes that must be written to storage and interfere with concurrent, synchronous reads. If the system resources are under-utilized, the server encourages clients to flush their dirty data by decreasing its price.

2.2 Asynchronous Reads

CA-NFS attempts to optimize the scheduling of asynchronous reads (read-ahead). Servers set the price for read-aheads based on the disk and network utilization. If the server resources are heavily congested, CA-NFS servers are less willing to accept read-ahead operations.

A client's willingness to perform read-ahead depends on its available memory and the effectiveness of the operation. If the server and network resources are congested so that the server's read-ahead price is higher than their local price, clients perform read-ahead prudently in favor of synchronous operations. Capping the number of read-ahead operations saves client memory, delays the consumption of network bandwidth, but often converts cache hits into synchronous reads because data were not preloaded into the cache. On the other hand, if the server price is low, clients perform read-ahead more aggressively.

2.3 CA-NFS in Practice

Figure 1 shows the high-level operation of the system and how the pricing model make clients adapt their behavior based on the state of the system. At this time, our treatment of pricing is qualitative. We describe the details of constructing appropriate pricing models in Section 3.3.

The server sets the price of different operations to manage its resources and network utilization in a coordinated fashion. In this example, the server's memory is near occupancy and it is near its maximum rate of I/O per second (IOPS). Based on this, it sets the price of asynchronous writes to be relatively high, because they consume server memory and add IOPS to the system.

CA-NFS allows the system to exchange memory consumption between the clients and the server. Clients adapt their prices based on their local state. Client #1 has available memory, so it stops writing dirty data. Client #2 is nearing its memory bound and, if it runs out of memory, applications will block awaiting the completion of asynchronous writes. Thus, even though the server price of asynchronous writes is high, this client is willing to

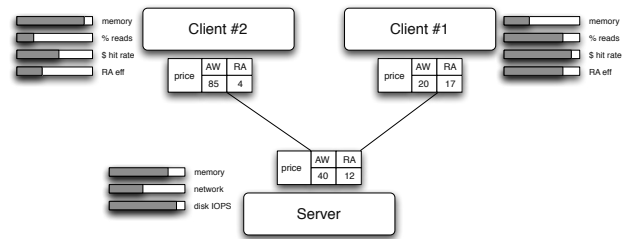


Figure 1: Overview of Congestion-Aware NFS. Clients and servers monitor their resource usage from which they derive prices for the different file system operations. (AW = asynchronous write, RA = read ahead, RA eff = read-ahead effectiveness.)

pay in order to avoid exhausting its memory. When the server clears its memory, it will lower the price of asynchronous writes and Client #1 will commence writing again. Servers notify clients about their prices as part of the CA-NFS protocol.

The criteria for whether to perform read-ahead prudently or aggressively are similar. Client #1 has lots of available memory, a read-dominated workload, and good read-ahead effectiveness, so that read-ahead turns most future synchronous reads into cache hits. Thus, it is willing to pay the server's price and perform more aggressive read-ahead. Client #2 has a write-dominated workload, little memory, and a relatively ineffective cache. Thus, it halts read-ahead requests to conserve resources for other tasks.

3 Pricing Mechanism

In distributed file systems, resources are heterogeneous and, therefore, no two of them are directly comparable. One cannot balance CPU cycles against memory utilization or vice versa. Nor does either resource convert naturally into network bandwidth. This makes the assessment of the load on a distributed system difficult. Previous models [20, 38, 44] designed to manage load and avoid throughput crashes via adaptive scheduling focus on one resource only or rely on high-level observations, such as request latency. The price unification model in CA-NFS provides several advantages: (a) it takes into account all system resources, (b) it unifies congestion across all devices in order to be comparable, and (c) it identifies bottlenecks across all clients and the server in a collective way.

Underlying the entire system, we develop a unified algorithmic framework based on competitive analysis for the efficient scheduling of distributed file system operations with respect to system resources. We rely on the algorithm of Awerbuch *et al.* [4] for bandwidth sharing in circuit-sharing networks with permanent connec-

tions that uses an online auction model to price congestion in a resource independent way. We adapt this theory to distributed file systems by considering the path of file system operations, from the client's memory to server's disk, as a short-lived circuit.

CA-NFS uses a reverse auction model. In a reverse auction, the buyer advertises a need for a service and the sellers place bids, like a regular auction. However, the seller who places the lowest bid wins the auction. Accordingly in CA-NFS, when the client is about to issue a request, it compares its local price with the server price. Depending on who offers the lower price the client accelerates, or defers the operation.

We start by describing an auction for a single resource. We then build a pricing function for each resource and assemble these functions into a price for each NFS operation.

3.1 Algorithmic Foundation

For each resource, we define a simple auction in an online setting in which the bids arrive sequentially and unpredictably. In a way, a bid represents the client's willingness to pay for the use of the resource, *i.e.* the client's local price. A bid will be accepted immediately if it is higher than the price of the resource at that time.

Our goal is to find an online algorithm that is competitive to the optimal offline algorithm in any future request sequence. The performance degradation of an online algorithm (competitive ratio) is $r = \max(B_{\text{offline}}/B_{\text{online}})$ in which B_{offline} is the benefit from the offline optimal algorithm and B_{online} the benefit from the online algorithm. Awerbuch *et al.* [4] establish the lower bound at $\Omega(\log k)$ in which k is the ratio between the maximum and minimum benefit realized by the online algorithm over all inputs. The lower bound is achieved when reserving $1/\log k$ of the resource doubles the price.

The worst case occurs when the offline algorithm sells the entire resource at the maximum bid P , which was rejected by the online algorithm. For the online algorithm to reject this bid, it must have set the price greater than P , which means it has already sold $1/\log k$ of the resource for at least $P/2$.

$$\begin{aligned} B_{\text{online}} &> \frac{P}{2 \log k} \quad \text{and} \\ B_{\text{offline}} - B_{\text{online}} &< P \\ \implies r &< 1 + 2 \log k \end{aligned}$$

Increasing price exponentially with increased utilization leads to a competitive ratio logarithmic in k .

3.2 A Practical Pricing Function

This model gives us an online strategy that is provably competitive with the optimal offline algorithm in the maximum usage of each resource. It has a weak (log, not constant) competitive ratio, but even this weak ratio is unprecedented in the storage system's literature. The online algorithm knows nothing about the future, assumes no correlation between past and future requests, and is only aware of the current system state.

Based on the theoretical framework, we define the pricing function P_i for an individual resource i in our framework as

$$P_i(u_i) = P_{\max} \frac{\{k_i^{u_i} - 1\}}{\{k_i - 1\}}$$

in which the utilization u_i varies between 0 and 1 so that the price varies between 0 and P_{\max} .

The parameter k represents the performance degradation experienced by the end user as the resource becomes congested. Thus, appropriate values of k should provide incremental feedback as the resource usage increases.

The heterogeneous resources of distributed file systems complicate parameter selection. Different resources become congested at different levels of utilization, which dictates that parameters need to be set individually. With very large k , the price function stays near zero until the utilization is almost 1. Then the price goes up very quickly. With very small k , the resource becomes expensive at lower utilization, which throttles usage prior to congestion. The network exhibits few negative effects from increased utilization until near its capacity and, thus, calls for a higher setting of k . Similarly, memory works well until it's nearly full at which point it experiences congestion in the form of fragmentation and synchronous stalls from out-of-memory conditions. Disks, on the other hand, require smaller values of k , because each additional I/O interferes with all subsequent (and some previous) I/Os, increasing the service time by increasing queue lengths and potentially moving the head out of position.

CA-NFS users do not need to set the value of k explicitly, as it is precomputed for most existing device types. The pricing mechanism is robust to small hardware variations, *e.g.* to different device brands. During various CA-NFS deployments, we experimented extensively with the value of k . (We do not present all these experiments as they are quite tedious.)

We approximate the cumulative cost of all resources by the highest cost (most congested) resource. The highest cost resource corresponds well with the system bottleneck. P_{\max} is the same for all server resources and the exponential nature of the pricing functions ensures that resources under load become expensive quickly.

In order to avoid the effects of over-tuning and enforce stability, we set two additional constraints on the cost function. Clients assign an infinitesimally higher value to the maximum price for their resources ($P_{max} + \epsilon$) than do servers. This ensures that when both the client and the server are overloaded, the client sends the operations to the server. In practice, servers deal with overload more gracefully than do clients. Also, the client's prices are always higher than a minimum price P_{min} so that if neither the client nor the server is congested, operations are performed at the server.

3.3 Calculating Resource Utilization

The theoretical model does not make any explicit assumptions about the type of resources managed. As a result, adding new resources to the system is straightforward. We currently monitor the effective usage of five resources, each with its own intricacies:

Server CPU: It is straightforward to establish the utilization of the CPU accurately at any given time through system monitoring.

Client and Server Network: The utilization of networks is also well defined. However, network bandwidth needs to be time-averaged to stabilize the auction. Without averaging, networks fluctuate between utilization 0 when idle and 1 when sending a message. The price would be similarly extreme and erratic. Thus, we monitor the average network bandwidth over a few hundreds of milliseconds.

Server Disk: Measuring disk utilization is difficult because of irregular response times. Although observed throughput seems a natural way to represent utilization, it is not practical because it depends heavily on the workload. A sequential workload experiences higher throughput than a random set of requests. However, disk utilization may be higher in the latter case, because the disk spends head time seeking among the random requests.

We measure disk utilization by sampling the length of the device's dispatch queue at regular, small time intervals. The maximum disk utilization depends on the system configuration. We do not identify the locality among pending operations nor do we use device-specific information. Recently, Fahrad [36] and Zygaris [21] showed the effectiveness of measuring disk utilization by examining the disk head time. We plan to evaluate this approach in future work.

Client and Server Memory: Pricing memory consumption is exceedingly difficult, because memory is a single resource used by many applications for many purposes, caching for reuse, dirty buffered pages, and read ahead. A cache must preserve a useful population of read-cache pages. Deferring writes in CA-NFS could reserve more

memory pages to buffer writes, which may in turn reduces cache hit rates. To avoid this, we identify the portion of RAM that is actively used to cache read data and the effectiveness of that cache. We then use pricing to preserve that portion of memory in order to maintain cache hit rates. The price of memory increases if the existing set of pages yields a high cache hit rate or there are a large number of dirty pages that have triggered write-back.

Previous research [6] allows us to effectively track the utility of read cache pages through the use of two ghost caches. We introduce a virtual resource to monitor by using the distribution of read requests among the ghost caches to calculate the projected cache hit rates, and thus, the effective memory utilization. A large fraction of read requests falling in these regions indicates that the client would benefit from more read caching, so deferring writes is not of particular benefit.

Client Read-Ahead Effectiveness: We define a virtual resource that captures the expected efficiency of read-ahead [24, 37]. We build our metric of read-ahead confidence on the adaptive read-ahead logic recently introduced in the Linux kernel [12]. We define confidence as the ratio of accesses to read-ahead pages divided by the total number of pages accessed for a specific file. For high values, the system performs read-ahead more aggressively. For low values, the kernel will be more reluctant to do the next read-ahead.

3.4 CA-NFS Implementation

We have implemented CA-NFS by modifying the existing Linux NFS client and server in the 2.6.18 kernel. Specifically, we added support for the exchange of pricing information and we changed the NFS write operation to add support for acceleration and deferral. We have also made modifications to the Linux memory manager to support the classification of the memory accesses and the read-ahead heuristics.

The CA-NFS server advertises cost information to clients, which implement the scheduling logic. We have overridden the FSSTAT protocol operation (NFSv3) to include pricing information about server resources. Normally, FSSTAT retrieves volatile file system state information, such as the total size of the file system or the amount of free space. Upon a client's FSSTAT request, the server encodes the prices of operations based on its monitored resource usage. In our implementation, the server computes the statistics of the resource utilization and updates its local cost information every one second. FSSTAT is a lightweight operation that adds practically no overhead to the system resources. Clients do not block waiting for the operation to complete.

Clients send an FSSTAT request to the server every ten READ or WRITE requests or when the time interval

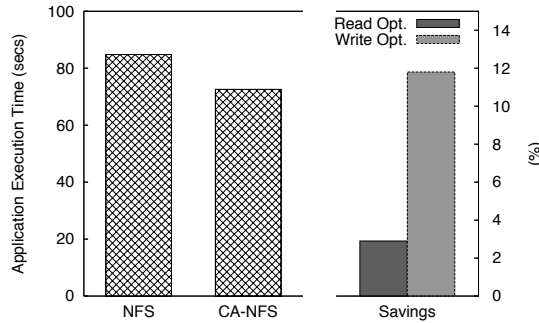


Figure 2: Time to copy a 4GB directory over NFS and CA-NFS and a breakdown of CA-NFS savings

from the previous query is more than ten seconds. As part of CA-NFS extensions, we intend to have the server notify active clients via callbacks when its resource usage increases sharply.

4 Evaluation

We run experiments on a cluster of twenty-four machines running at 3.2GHz with 2GB of RAM each. One machine has 4GB of RAM and acts as the server. All nodes are connected via Gigabit Ethernet. To compare CA-NFS with NFSv3, we run a set of micro-benchmarks and application workloads based on the different profiles available in *filebench* [25], Sun’s filesystem benchmark, and *IOzone* [18].

4.1 Microbenchmarks

We start our analysis with a simple *filebench* experiment. A single thread of just one client copies a large directory of 4GB over CA-NFS and NFS. This workload creates a hierarchical directory tree, then measures the rate at which files can be copied from the source tree to the new tree. The sizes of the files in the directory vary from 1KB to 200MB. Even in such a simple configuration, CA-NFS provides 15% improvement in performance, measured by completion time (Figure 2).

Regular NFS clients fail to use their local memory to good effect even though it is not congested. NFS clients read data from the server and start buffering write pages until they reach the statically defined limit of dirty pages. Then, the flushing daemon forces the pages to be written to the server. This requires the server to harden data to disk. The resulting write traffic delays disk read requests. In contrast, CA-NFS clients determine that the server disk is heavily utilized through the exchange of pricing information. CA-NFS clients use a much larger portion of their RAM to buffer dirty pages, avoiding the large, asynchronous writes to the server that interfere with reads. The effects of read-ahead optimizations are

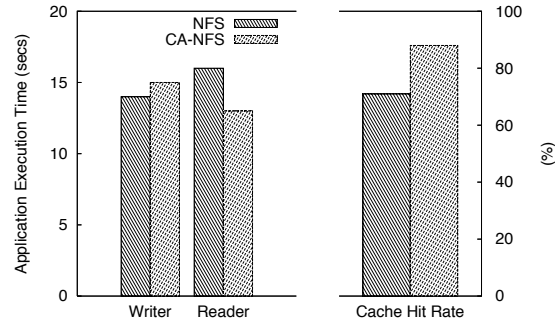


Figure 3: Application execution time and cache hit rates when accelerating writes

less dramatic, but still important. Read-aheads are issued aggressively in the beginning, because there is free memory space and they yield a high hit rate for this mostly sequential workload. As the server memory resources become more congested with dirty pages, client-initiated read-aheads are performed more prudently. Figure 2 also breaks out the portion of the improvement attributed to write (12%) and read-ahead optimizations (3%).

4.1.1 Operation Scheduling

Accelerating writes: The next experiment combines two *IOzone* workloads to show how CA-NFS preserves cache hit rates by valuing client memory highly. We consider a client application that writes a 2GB file sequentially. On the same client, another application performs re-reads, i.e. reads that will be server cache hits if the system does not evict the pages.

Figure 3 shows the execution times of the two applications for NFS and CA-NFS. CA-NFS improves read performance by 21% when compared with NFS. The NFS client evicts memory pages used for read caching in order to buffer writes. This reduces the cache hit rate and application-perceived read performance as a consequence. NFS clients replace approximately 15% of the pages used for caching and realize a cache hit rate of only 70%. In contrast, CA-NFS accelerates writes by flushing them immediately, anticipating the importance of the cache contents. CA-NFS clients maintain a cache hit rate of 90%. The client prefers to accelerate all asynchronous writes, because its read cache is producing a high hit rate, thus its price for asynchronous writes is high. The server price for asynchronous writes is low, because none of its resources is congested.

Deferring writes: We now demonstrate how CA-NFS uses write-buffering at the client to avoid I/O interference at the server. One client issues random reads that are serviced by the server’s disk. Another client writes a 1GB file to the server. The NFS client sends the write requests

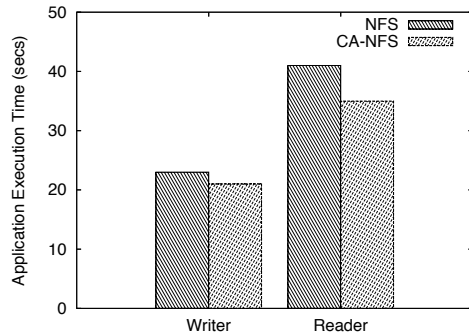


Figure 4: Execution time for two clients reading and writing to a file when deferring writes

to the server, which flushes them to stable storage. These disk writes increase the service time of disk reads, because they interfere at the disk with read requests coming from the first client. Through pricing, CA-NFS identifies congestion at the disk, which causes the writing client to buffer dirty data and reduces the amount of write data delivered to the server. Figure 4 shows that CA-NFS improves read performance by 18%. In this case, write performance is also improved by 6%.

4.1.2 On the Pricing Metric

We characterize how the pricing function captures system dynamics by comparing resource utilization and resource price side-by-side. We show that pricing reflects congestion on heterogeneous resources, i.e. on networks, for memory, and on the disk. Prices create a single view of system load in a resource independent manner.

For the network resource, we run a network intensive workload with four clients reading a 1GB file from server's memory at a rate of 50MB/sec each over a GbE network. Three clients suffice to saturate the network bandwidth. We start each client at 10 second time intervals in order to provide incremental load to the system. Figure 5(a) plots the server-perceived throughput and the average throughput at the clients. Figure 5(b) shows the server's system price, governed by the network, at the same time scale. As the system load increases, each client gets a smaller share of the bandwidth and average client throughput drops. Over time, the network price increases to near its maximum value (1.0, the value of P_{max} in all experiments). The increase in price causes the clients to back off, preventing overload, and the server throughput remains stable under heavy load.

We run a similar experiment for memory-bound workloads and memory price. A client issues reads by increasing the number of requests for already accessed data. By recycling the client's memory, we force all re-read requests to be serviced out of the server cache only. From the server's perspective, as the hit ratio increases cache

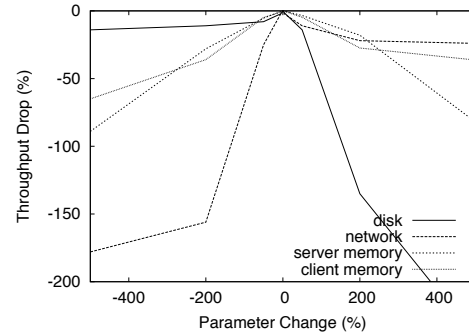


Figure 6: CA-NFS throughput sensitivity to the correct parameterization of k

data become more important, so it increases the price for memory operations (Figures 5(c) and 5(d)).

For disk-bound workloads, we run a client process that issues random read requests over increasingly larger spans of the disk. As we increase the span, client throughput drops from increased disk head utilization that leads to more requests in the disk dispatch queue (Figure 5(e)). In response to the increase in the number of pending requests, the system increases the price for the disk resource (Figure 5(f)).

In the next experiment, we show how the selection of parameter k affects system performance. We run a read-write *IOzone* workload on two clients accessing a 2GB file on the server. Through measurements, we have established a value of k for each device type, which yields the best throughput for this experiment. We alter the value of k for the client and server memory, the network and the disk resources, and we examine the drop in system throughput.

Figure 6 shows that small perturbations of k do not affect CA-NFS performance. However, if the value of k differs significantly from its optimal (as calculated) setting, performance degradation is notable. Low values of k lead to underutilization of the system resources, while high values make the system less adaptive, as prices increase very rapidly. Figure 6 also shows that the disk and the network are more sensitive to correct parameterization. This is because, these resources exhibit very high (disk) or very low interference (network) between past and future requests. As already mentioned, CA-NFS users do not have to set the value of k explicitly. In the next set of experiments, we show that the CA-NFS parameter selection is robust to different workload types.

4.2 Application Benchmarks

Microbenchmark experiments demonstrate the operation of CA-NFS by isolating the benefits of individual optimizations. To better understand how CA-NFS effects applications, we turn our attention to macrobenchmarks.

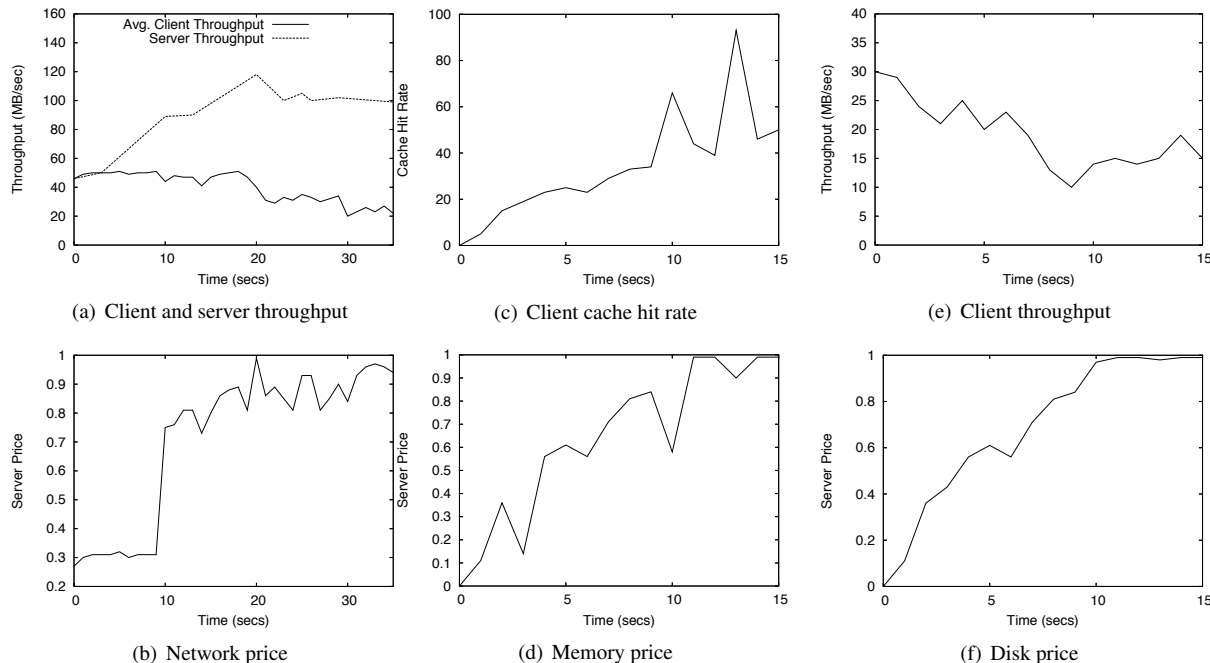


Figure 5: Examining the pricing mechanism for three different resources (network (a,b), memory (c,d), and disk (e,f))

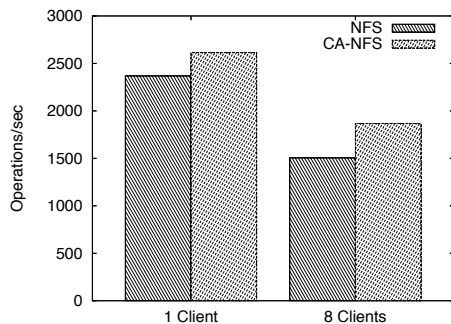


Figure 7: Average number of ops/sec per client for the file-server benchmark

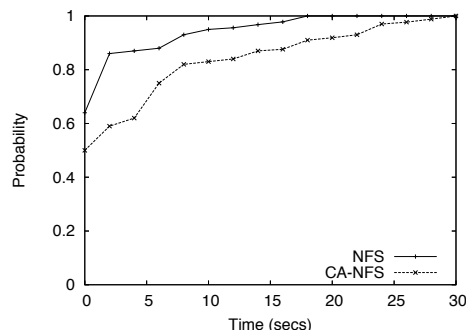


Figure 8: CDF of the time that the system schedules write-backs for NFS and CA-NFS

First, we evaluate CA-NFS by running the `fileserver` synthetic workload provided by `filebench`, on eight clients. This workload is modeled after SPECsfs [39], an industry standard test suite that is based on data collected by SFS committee members from thousands of real NFS servers operating at customer sites. The test performs a sequence of creates, deletes, appends, reads, writes and attribute operations on the file system. We randomly set the number of user threads, the number of files written and the average file size to numbers between 100-200, 1000-5000 and 100-5120KB respectively. This workload contains a large number of asynchronous operations.

Figure 7 shows that CA-NFS outperforms NFS by more than 10% in the single client setup and by more

than 20% in the eight-client setup. Figure 8 shows the cumulative distribution function of the time that elapses between a write operation submitted by the application and the relevant pages marked for commit by the file system. CA-NFS schedules asynchronous write operations very differently from NFS. NFS clients are forced to commit many pages almost immediately as they become dirty, in order to prevent the system from running out of memory to buffer dirty pages. No page stays dirty for more than 12 seconds after the write is issued. CA-NFS schedules the write-back operations more evenly across the 30-second time frame that defines the reliability window for asynchronous writes in most current operating systems. As a result, traffic in CA-NFS is less bursty, a significant factor that improves performance.

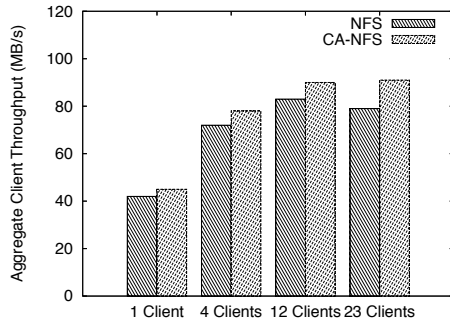


Figure 9: Aggregate client throughput for the oltp benchmark a function of the number of clients

	NFS	CA-NFS
ops/sec	2443	2503
ms/op		
open file	34.3	33.7
read file	5.3	5.1
close file	4.0	4.2
append log	7.1	7.2

Table 1: NFS and CA-NFS under the webserver workload

The next benchmark examines CA-NFS characteristics under an OLTP workload that performs transactions into a filesystem using an I/O model from Oracle 9i. This workload tests for the performance of small random reads and writes in conjunction with moderate (128KB) synchronous writes. Operations represent read and write OLTP transactions and writes to the log file respectively. On each client, we launch 200 reader processes, 10 processes for asynchronous writing, and a log writer. We run the experiments four times, modifying the number of active clients.

For the `oltp` workload, CA-NFS is more scalable than NFS. Although this workload exhibits some cache locality on the server, the main bottleneck in this experiment is the server’s disk, which is overwhelmed by the number of incoming requests. Figure 9 plots the aggregate client throughput for different client populations. For a small number of clients (one to four), CA-NFS provides a rather small performance advantage. As the number of clients increases, the relative throughput of CA-NFS increases when compared with NFS. In the case of NFS, the aggregate throughput for the 23-client setup is less than in the 12-client setup. This is because the number of incoming requests overwhelms the server resulting in a throughput crash.

In our last experiment, we examine the performance of CA-NFS under a workload that contains mostly syn-

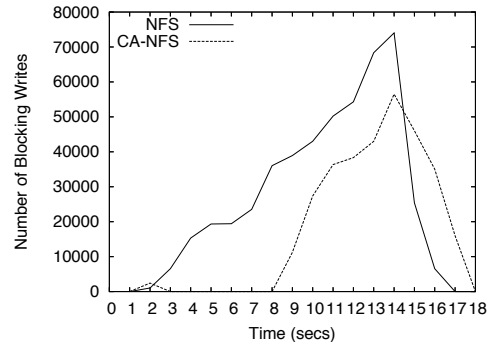


Figure 10: Number of asynchronous client writes blocked

chronous read operations. The file server exports its filesystem to a number of web servers. The `webserver` workload from the *filebench* suite consists of a mix of open/read/close of multiple files in a directory tree, plus a file append (to simulate the web log) in which 16KB is appended to the weblog for every 10 reads.

CA-NFS performs slightly better (about 3%) than NFS thanks to the read-ahead optimizations. Write optimizations are not a factor in this benchmark, because of the small number of write operations. Table 4.2 shows that for all operations the latency for both NFS and CA-NFS is almost identical.

Macrobenchmark experiments show that CA-NFS significantly outperforms NFS under workloads with a significant number of asynchronous operations, such as the `fileserver` benchmark. For workloads that are read-dominated (`webserver`) or contain small, asynchronous requests (`oltp`), CA-NFS performs comparably to NFS, showing that our modifications are lightweight.

4.3 High-Speed Hazards

To further evaluate our framework, we perform measurements over a 10-Gbps Infiniband network. As opposed to the previous set of experiments, in this setup, the network bandwidth outstrips disk transfer rates. We consider the two clients writing file data sequentially to the server for fifteen seconds over NFS and CA-NFS. During the write burst, both clients write data at the maximum rate, close to 200MB/sec.

This experiment shows that running out of memory turns asynchronous file system operations into synchronous that block all progress (Figure 10). Regular NFS experiences synchronous waits for asynchronous writes starting at 2 seconds. When the number of dirty pages on the NFS clients reaches the flushing point, clients start writing data, which overwhelms the disk system and memory available to buffer writes at the server fills. All subsequent writes block awaiting completion.

CA-NFS detects congestion on the server memory and I/O system through pricing and buffers writes in local memory. This makes the effective write-buffering space 8GB, 2GB on each client and 4GB on the server, rather than the 4GB of server memory that NFS uses. CA-NFS does not experience synchronous waits until 8 seconds and blocks fewer writes overall. This results in higher overall throughput as well. CA-NFS writes 4.8GB worth of data whereas regular NFS writes only 3.9GB, an improvement of 23%.

This scenario shows how the emergence of high-speed networking makes holistic storage management critical. For storage systems, we are on the verge of a new era. Infiniband and 10Gbps Ethernet deliver data at such rapid rates that storage systems that receive and process these data cannot keep up. This gap between network bandwidth and disk throughput creates a memory crisis for storage servers. Many clients writing data in parallel will create a data stream that a server cannot transfer to disk. Flow control in the transport protocol will be irrelevant, because the system is not network bound. The server buffers data pending I/O completion and the buffered data accumulate until memory is full. This results in a cascading throughput crash over the entire system [11].

5 Future Directions

Although our focus is on the scheduling of asynchronous operations, pricing synchronous operations wisely can enable the system to manage nonstandard I/O processes. Distributed file systems often have lower-priority I/O tasks, such as data mining, indexing, backup, etc. Capping the willingness to pay for synchronous operations causes these low-priority tasks to halt automatically when resources become congested. Clients can also encode application priorities and differentiate between critical and noncritical tasks by charging different processes different prices.

The proposed framework does not address the issue of fairness over time. Operation costs are proportional to the current state of the system but independent of the client that put the system into the state. For example, one client could fill the server cache with dirty data, pushing up prices for all others.

Finally, more complex resource management goals can be realized by adding constraints to the auction model. For example, resource reservations can be accomplished by differentially pricing the same resource among clients. The goal is to insulate one client from the consumption by non-reserving clients. To do so, we need to limit the spending of non-reserving clients and increase resource prices prior to exhaustion, creating an artificial shortage. Also, proportional sharing arises when clients are given salaries, i.e. a rate of consump-

tion or fixed amount of spending over some time interval. This concept extends the ideas of flow control beyond networks to cover all resources in the system. Pricing certain resources and making all other resources free, allows sharing to be targeted to specific resources only.

6 Related Work

Economic Models: Using economic models for resource management is not a novel approach [10]. Auction-based systems have been applied in a broad range of distributed systems including clusters [9], computational grids [26], parallel computers [40], and Internet computing systems [27]. These systems are intended for coarse-grained resource allocations.

Network Flow Control: Flow control schemes offer to each client a proportional share of the network and, thus, guarantee to a large extent fairness [31]. Many different approaches exist in the literature, including TCP-like window based protocols [14, 19], feedback schemes [13], and optimization based methods [15].

The congestion pricing techniques upon which we build have been used by Amir *et al.* [2] to manage a single network resource. Kelly [23] was the first to describe pricing for flow and congestion control. However, our approach and Amir's are algorithmic, whereas Kelly relies on economic theory.

Memory Management: Li *et al* [28] acknowledge the asynchronous nature of writes and their dependence on the client's state. They propose a scheme where the storage clients inform the storage servers about the types of writes that they perform by passing write hints. These write hints can then be used by the server to manage the second-tier cache.

Carson and Setia [7] showed that for many workloads, periodic updates from a write-back cache perform worse than write-through caching. They suggest two alternate disciplines: (1) giving reads non-preemptive priority and (2) interval periodic writes in which each write gets its own fixed period in the cache. Mogul [32] implements an approximate interval periodic write-back policy that staggers writes in time using a small (one second) timer. Golding *et al* [16] delay write-back until the system reaches an idle period. This reduces the delays seen by reads by postponing competing writes until idle periods, possibly with the help of nonvolatile memory, in order to ensure consistency.

Storage controllers with nonvolatile memory employ adaptive destaging policies that vary the rate of writing [1, 43] or the destage threshold [33, 43], based on memory occupancy and filling and draining rates. In these systems, cached writes are persistent, so they want to delay destaging data as long as possible.

Patterson *et al* [35] in TIP made cache residency and prefetching decisions over the network following a cost benefit analysis. Their work was based on models that value memory pages for different type of data, such as prefetched, buffered, or cached. Nelson *et al* [34] in Sprite mentioned a weight used to trade off how to partition memory between pages for the file cache and for virtual memory. Nelson's principle was not applied to a distributed context. These approaches use heuristic methods and do not look at the relative load across all clients.

Storage Scheduling and QoS: Storage quality of service (QoS) attempts to optimize the system resources individually [17, 29] or conjunctively [22]. Fairness in the QoS context is generalized to incorporate weights used to introduce deliberate bias, depending for example on different service-level agreements (SLAs) [45].

In general, quality of service (QoS) approaches are not well-suited for multi-resource optimization. CA-NFS complements QoS methods [8, 22, 30, 42] that employ I/O throttling in order to limit resource congestion and avoid throughput crashes. We do not offer the performance guarantees to applications on which one might build SLAs [29]. Instead, we follow a best-effort approach to improve application-perceived performance by minimizing latency and maximizing throughput for synchronous file system operations.

Provisioning: Provisioning systems use a single metric, utility or cost in dollars, to unify heterogeneous resources when deciding the initial configuration of a system under a fixed utility budget. Recently, Strunk *et al.* [41] provide a framework for provisioning based on detailed system models and genetic algorithms to explore the configuration space. This extends the previous work on provisioning of Anderson *et al.* [3].

While the unification of resources using utility is superficially similar to pricing, provisioning solves a very different problem. Provisioning determines how to achieve the best availability, throughput, or IOPS under a fixed budget as a static offline configuration problem. CA-NFS examines dynamic pricing of operations under changing workloads in static configurations.

7 Conclusions

We have shown the importance of using holistic performance management for the adaptive scheduling of lower-priority distributed file system requests based on system congestion in order to reduce their interference with foreground, synchronous requests. We also show the virtue of adaptation based on application-perceived performance, rather than server-centric metrics.

CA-NFS introduces a new dimension in resource management by implicitly managing and coordinating the us-

age of the file system resources among all clients. It unifies fairness and priorities in a single framework that assures that realizing optimization goals will benefit file system users, not the file system servers.

Acknowledgements

We would like to thank the NetApp Kilo-Client support team, and especially Alan Belanger, for providing the environment in which we tested CA-NFS. Also, we thank our shepherd, Narasimha Reddy, for all of his help. This work was supported in part by NSF awards IIS-0456027 and CCF-0238305.

References

- [1] M. Alonso and V. Santonja. A new destage algorithm for disk cache: DOME. In *EUROMICRO Conference*, 1999.
- [2] Y. Amir, B. Awerbuch, C. Danilov, and J. Stanton. A cost-benefit flow control for reliable multicast and unicast in overlay networks. In *IEEE/ACM Transactions on Networking*, 2005.
- [3] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 2005.
- [4] B. Awerbuch, Y. Azar, S. A. Plotkin, and O. Waarts. Competitive routing of virtual circuits with unknown duration. In *Symposium on Discrete Algorithms*, 1994.
- [5] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *ACM Symposium on Operating Systems Principles*, 1991.
- [6] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey. AWOL: An adaptive write optimizations in layer. In *Conference on File and Storage Technologies*, 2008.
- [7] S. D. Carson and S. Setia. Analysis of the periodic update write policy for disk cache. *IEEE Transactions on Software Engineering*, 18(1), 1992.
- [8] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance virtualization for large-scale storage systems. *Symposium on Reliable Distributed Systems*, 2003.
- [9] B. N. Chun and D. E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *CC-GRID '02: IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.
- [10] S. Clearwater. Market-based control: A paradigm for distributed resource allocation. *World Scientific*, 1996.
- [11] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Symposium on Operating Systems Design and Implementation*, 1996.
- [12] W. Fengguang. Adaptive read-ahead in the Linux kernel. <http://lwn.net/Articles/155097/>.

- [13] S. Floyd. TCP and explicit congestion notification. *ACM Computer Communication Review*, 24(5):10–23, 1994.
- [14] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.
- [15] R. Gibbens and F. Kelly. Resource pricing and the evolution of congestion control. *Automatica*, 1999.
- [16] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness Is Not Sloth. In *USENIX Annual Technical Conference*, 1995.
- [17] P. Goyal, D. Jadav, D. S. Modha, and R. Tewari. CacheCOW: QoS for storage system caches. In *International Workshop on Quality of Service (IWQoS 03)*, 2003.
- [18] The IOzone Benchmark. <http://www.iozone.com>.
- [19] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM*, 1988.
- [20] M. B. Jones, D. Rou, and M. Rou. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Symposium of Operating Systems and Principles*, 1997.
- [21] T. Kaldewey, T. Wong, R. Golding, A. Povzner, S. A. Brandt, and C. Maltzahn. Virtualizing disk performance. In *Real-Time and Embedded Technology and Applications Symp*, 2008.
- [22] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems, 2004.
- [23] F. Kelly, A. Maulloo, and D. Tan. Rate control in communication networks: Shadow prices, proportional fairness and stability. In *Journal of the Operational Research Society*, volume 49, 1998.
- [24] A. Ki and A. E. Knowles. Adaptive data prefetching using cache information. In *International Conference on Supercomputing*, 1997.
- [25] E. Kustarz, S. Shepler, and A. Wilson. The new and improved filebench file system benchmarking framework. *Conference on File and Storage Technologies*, 2008.
- [26] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiaгент Grid Syst.*, 2005.
- [27] L. Levy, L. Blumrosen, and N. Nisan. On line markets for distributed object services: the majic system. In *USITS'01: USENIX Symposium on Internet Technologies and Systems*, 2001.
- [28] X. Li, A. Aboulmaga, K. Salem, A. Sachendina, and S. Gao. Second-tier cache management using write hints. In *Conference on File and Storage Technologies*, 2005.
- [29] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework for qos guarantees and its application to differentiated caching services. In *International Workshop on Quality of Service*, 2002.
- [30] C. Lumb, A. Merchant, and G. Alvarez. Facade: Virtual storage devices with performance guarantees. In *Conference on File and Storage Technologies*, 2003.
- [31] L. Massoulie and J. Roberts. Bandwidth sharing: Objectives and algorithms. In *INFOCOM*, 1999.
- [32] J. Mogul. A better update policy. In *USENIX Summer Technical Conference*, 1994.
- [33] Y. J. Nam and C. Park. An adaptive high-low watermark destage algorithm for cached RAID5. In *Pacific Rim International Symposium on Dependable Computing*, 2002.
- [34] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the sprite network file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.
- [35] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *ACM Symposium on Operating Systems Principles*, 1995.
- [36] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. *SIGOPS Oper. Syst. Rev.*, 42(4):13–25, 2008.
- [37] D. Revel, D. McNamee, D. Steere, and J. Walpole. Adaptive prefetching for device independent file I/O. Technical Report CSE-97-005, Oregon Graduate Institute School of Science and Engineering, 1997.
- [38] A. Riska, E. Riedel, and S. Iren. Adaptive disk scheduling for overload management. In *International Conference on the Quantitative Evaluation of Systems*, 2004.
- [39] The SPEC SFS97R1 (3.0) Benchmark. Available at <http://www.spec.org/sfs97r1>.
- [40] I. Stoica, H. Abdel-Wahab, and A. Pothen. A Microeconomic Scheduler for Parallel Computers. In *Workshop on Job Scheduling Strategies for Parallel Processing*, 1995.
- [41] J. D. Strunk, E. Theresecka, C. Faloutsos, and G. R. Ganger. Using utility to provision storage systems. In *Conference on File and Storage Technologies*, 2008.
- [42] S. Uttamchandani, L. Yin, G. A. Alvarez, J. Palmer, and G. Agha. Chameleon: a self-evolving, fully-adaptive resource arbitrator for storage systems. In *USENIX Annual Technical Conference*, 2005.
- [43] A. Varma and Q. Jacobsen. Destage algorithms for disk arrays with nonvolatile caches. *IEEE Transactions on Computers*, 47(2), 1995.
- [44] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned scalable internet services. In *Symposium of Operating Systems Principles*, 2001.
- [45] Z. Zimmermann and U. Killat. Resource marking and fair rate allocation. In *International Conference on Communications*, 2002.

Trademark Notice: NetApp, the NetApp logo, and Go further, faster are trademarks or registered trademarks and NetApp Inc in the U.S. and other countries. Linux is a registered trademark of Linus Torvalds. All other brands or products are trademarks or registered trademarks of their respective holders and should be treated as such.

Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality

Mark Lillibridge[†], Kave Eshghi[†], Deepavali Bhagwat[‡], Vinay Deolalikar[†], Greg Trezise[#],
and Peter Camble[#]

[†]*HP Labs*

[‡]*UC Santa Cruz*

[#]*HP Storage Works Division*

first.last@hp.com

Abstract

We present *sparse indexing*, a technique that uses sampling and exploits the inherent locality within backup streams to solve for large-scale backup (e.g., hundreds of terabytes) the chunk-lookup disk bottleneck problem that inline, chunk-based deduplication schemes face. The problem is that these schemes traditionally require a full chunk index, which indexes every chunk, in order to determine which chunks have already been stored; unfortunately, at scale it is impractical to keep such an index in RAM and a disk-based index with one seek per incoming chunk is far too slow.

We perform stream deduplication by breaking up an incoming stream into relatively large segments and deduplicating each segment against only a few of the most similar previous segments. To identify similar segments, we use sampling and a sparse index. We choose a small portion of the chunks in the stream as samples; our sparse index maps these samples to the existing segments in which they occur. Thus, we avoid the need for a full chunk index. Since only the sampled chunks' hashes are kept in RAM and the sampling rate is low, we dramatically reduce the RAM to disk ratio for effective deduplication. At the same time, only a few seeks are required per segment so the chunk-lookup disk bottleneck is avoided. Sparse indexing has recently been incorporated into number of Hewlett-Packard backup products.

1 Introduction

Traditionally, magnetic tape has been used for data backup. With the explosion in disk capacity, it is now affordable to use disk for data backup. Disk, unlike tape, is random access and can significantly speed up backup and restore operations. Accordingly, disk-to-disk backup (D2D) has become the preferred backup option for organizations [3].

Deduplication can increase the effective capability of

a D2D device by one or two orders of magnitude [4]. Deduplication can accomplish this because backup sets have massive redundancy due to the facts that a large proportion of data does not change between backup sessions and that files are often shared between machines. Deduplication, which is practical only with random-access devices, removes this redundancy by storing duplicate data only once and has become an essential feature of disk-to-disk backup solutions.

We believe chunk-based deduplication is the deduplication method best suited to D2D: it deduplicates data both across backups and within backups and does not require any knowledge of the backup data format. With this method, data to be deduplicated is broken into variable-length chunks using content-based chunk boundaries [20], and incoming chunks are compared with the chunks in the store by hash comparison; only chunks that are not already there are stored. We are interested in *inline* deduplication, where data is deduplicated as it arrives rather than later in batch mode, because of its capacity, bandwidth, and simplicity advantages (see Section 2.2).

Unfortunately, inline, chunk-based deduplication when used at large scale faces what is known as the *chunk-lookup disk bottleneck problem*: Traditionally, this method requires a full chunk index, which maps each chunk's hash to where that chunk is stored on disk, in order to determine which chunks have already been stored. However, at useful D2D scales (e.g., 10-100 TB), it is impractical to keep such a large index in RAM and a disk-based index with one seek per incoming chunk is far too slow (see Section 2.3).

This problem has been addressed in the literature by Zhu *et al.* [28], who tackle it by using an in-memory Bloom Filter and caching index fragments, where each fragment indexes a set of chunks found together in the input. In this paper, we show a different way of solving this problem in the context of data stream deduplication (the D2D case). Our solution has the advantage that it

uses significantly less RAM than Zhu *et al.*'s approach.

To solve the chunk-lookup disk bottleneck problem, we rely on *chunk locality*: the tendency for chunks in backup data streams to reoccur together. That is, if the last time we encountered chunk A, it was surrounded by chunks B, C, and D, then the next time we encounter A (even in a different backup) it is likely that we will also encounter B, C, or D nearby. This differs from traditional notions of locality because occurrences of A may be separated by very long intervals (e.g., terabytes). A derived property we take advantage of is that if two pieces of backup streams share any chunks, they are likely to share many chunks.

We perform stream deduplication by breaking up each input stream into segments, each of which contains thousands of chunks. For each segment, we choose a few of the most similar segments that have been stored previously. We deduplicate each segment against only its chosen few segments, thus avoiding the need for a full chunk index. Because of the high chunk locality of backup streams, this still provides highly effective deduplication.

To identify similar segments, we use sampling and a sparse index. We choose a small portion of the chunks as samples; our sparse index maps these samples' hashes to the already-stored segments in which they occur. By using an appropriate low sampling rate, we can ensure that the sparse index is small enough to fit easily into RAM while still obtaining excellent deduplication. At the same time, only a few seeks are required per segment to load its chosen segments' information avoiding any disk bottleneck and achieving good throughput.

Of course, since we deduplicate each segment against only a limited number of other segments, we occasionally store duplicate chunks. However, due to our lower RAM requirements, we can afford to use smaller chunks, which more than compensates for the loss of deduplication the occasional duplicate chunk causes. The approach described in this paper has recently been incorporated into a number of Hewlett-Packard backup products.

The rest of this paper is organized as follows: in the next section, we provide more background. In Section 3, we describe our approach to doing chunk-based deduplication. In Section 4, we report on various simulation experiments with real data, including a comparison with Zhu *et al.*, and on the ongoing productization of this work. Finally, we describe related work in Section 5 and our conclusions in Section 6.

2 Background

2.1 D2D usage

There are two modes in which D2D is performed today, using a network-attached-storage (NAS) protocol and us-

ing a Virtual Tape Library (VTL) protocol:

In the NAS approach, the backup device is treated as a network-attached storage device, and files are copied to it using protocols such as NFS and CIFS. To achieve high throughput, typically large directory trees are coalesced, using a utility such as tar, and the resulting tar file stored on the backup device. Note that tar can operate either in incremental or in full mode.

The VTL approach is for backward compatibility with existing backup agents. There is a large installed base of thousands of backup agents that send their data to tape libraries using a standard tape library protocol. To make the job of migrating to disk-based backup easier, vendors provide Virtual Tape Libraries: backup storage devices that emulate the tape library protocol for I/O, but use disk-based storage internally.

In both NAS and VTL-based D2D, the backup data is presented to the backup storage device as a stream. In the case of VTL, the stream is the virtual tape image, and in the case of NAS-based backup, the stream is the large tar file that is generated by the client. In both cases, the stream can be quite large: a single tape image can be 400 GB, for example.

2.2 Inline versus out-of-line deduplication

Inline deduplication refers to deduplication processes where the data is deduplicated as it arrives and before it hits disk, as opposed to out-of-line (also called post-process) deduplication where the data is first accumulated in an on-disk holding area and then deduplicated later in batch mode. With out-of-line deduplication, the chunk-lookup disk bottleneck can be avoided by using batch processing algorithms, such as hash join [24], to find chunks with identical hashes.

However, out-of-line deduplication has several disadvantages compared to inline deduplication: (a) the need for an on-disk holding area large enough to hold an entire backup window's worth of raw data can substantially diminish storage capacity,¹ (b) all the functionality that a D2D device provides (data restoration, data replication, compression, etc.) must be implemented and/or tested separately for the raw holding area as well as the deduplicated store, and (c) it is not possible to conserve network or disk bandwidth because every chunk must be written to the holding area on disk.

2.3 The chunk-lookup disk bottleneck

The traditional way to implement inline, chunk-based deduplication is to use a *full chunk index*: a key-value index of all the stored chunks, where the key is a chunk's hash, and the value holds metadata about that chunk, including where it is stored on disk [22, 14]. When an

incoming chunk is to be stored, its hash is looked up in the full index, and the chunk is stored only if no entry is found for its hash. We refer to this approach as the full index approach.

Using a small chunk size is crucial for high-quality chunk-based deduplication because most duplicate data regions are not particularly large. For example, for our data set Workgroup (see Section 4.2), switching from 4 to 8 KB average-size chunks reduces the deduplication factor (original size/deduplicated size) from 13 to 11; switching to 16 KB chunks further reduces it to 9.

This need for a small chunk size means that the full chunk index consumes a great deal of space for large stores. Consider, for example, a store that contains 10 TB of unique data and uses 4 KB chunks. Then there are 2.7×10^9 unique chunks. Assuming that every hash entry in the index consumes 40 bytes, we need 100 GB of storage for the full index.

It is not cost effective to keep all of this index in RAM. However, if we keep the index on disk, due to the lack of short-term locality in the stream of incoming chunk hashes, we will need one disk seek per chunk hash lookup. If a seek on average takes 4 ms, this means we can look up only 250 chunks per second for a processing rate of 1 MB/s, which is not acceptable. This is the chunk-lookup disk bottleneck that needs to be avoided.

3 Our Approach

Under the sparse indexing approach, *segments* are the unit of storage and retrieval. A segment is a sequence of chunks. Data streams are broken into segments in a two step process: first, the data stream is broken into a sequence of variable-length chunks using a chunking algorithm, and, second, the resulting chunk sequence is broken into a sequence of segments using a segmenting algorithm. Segments are usually on the order of a few megabytes. We say that two segments are similar if they share a number of chunks.

Segments are represented in the store using their *manifests*: a manifest or segment recipe [25] is a data structure that allows reconstructing a segment from its chunks, which are stored separately in one or more chunk containers to allow for sharing of chunks between segments. A segment's manifest records its sequence of chunks, giving for each chunk its hash, where it is stored on disk, and possibly its length. Every stored segment has a manifest that is stored on disk.

Incoming segments are deduplicated against similar, existing segments in the store. Deduplication proceeds in two steps: first, we identify among all the segments in the store some that are most similar to the incoming segment, which we call *champions*, and, second, we deduplicate against those segments by finding the chunks they

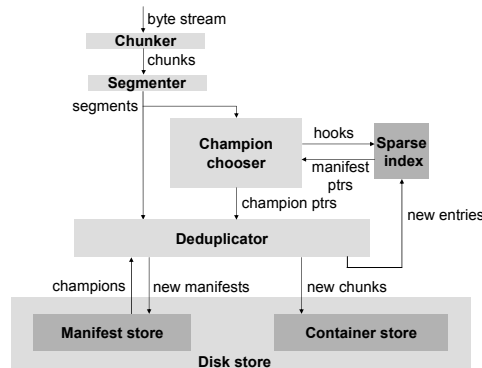


Figure 1: Block diagram of the deduplication process

share with the incoming segment, which do not need to be stored again.

To identify similar segments, we sample the chunk hashes of the incoming segment, and use an in-RAM index to determine which already-stored segments contain how many of those hashes. A simple and fast way to sample is to choose as a sample every hash whose first n bits are zero; this results in an average *sampling rate* of $1/2^n$; that is, on average one in 2^n hashes is chosen as a sample. We call the chosen hashes *hooks*.

The in-memory index, called the *sparse index*, maps hooks to the manifests in which they occur. The manifests themselves are kept on disk; the sparse index holds only pointers to them. Once we have chosen champions, we can load their manifests into RAM and use them to deduplicate the incoming segment. Note that although we choose champions because they share hooks with the incoming segment (and thus, the chunks with those hashes), as a consequence of chunk locality they are likely to share many other chunks with the incoming segment as well.

We will now describe the deduplication process in more detail. A block diagram of the process can be found in Figure 1.

3.1 Chunking and segmenting

Content-based chunking has been studied at length in the literature [1, 16, 20]. We use our Two-Threshold Two-Divisor (TTTD) chunking algorithm [13] to subdivide the incoming data stream into chunks. TTTD produces variable-sized chunks with smaller size variation than other chunking algorithms, leading to superior deduplication.

We consider two different segmentation algorithms in this paper, each of which takes a target segment size as a parameter. The first algorithm, *fixed-size segmentation*, chops the stream of incoming chunks just before the first

chunk whose inclusion would make the current segment longer than the goal segment length. “Fixed-sized” segments thus actually have a small amount of size variation because we round down to the nearest chunk boundary. We believe that it is important to make segment boundaries coincide with chunk boundaries to avoid split chunks, which have no chance of being deduplicated.

Because we perform deduplication by finding segments similar to an incoming segment and deduplicating against them, it is important that the similarity between an incoming segment and the most similar existing segments in the store be as high as possible. Fixed-size segmentation does not perform as well here as we would like because of the *boundary-shifting problem* [13]: Consider, for example, two data streams that are identical except that the first stream has an extra half-a-segment size worth of data at the front. With fixed-size segmentation, segments in the second stream will only have 50% overlap with the segments in the first stream, even though the two streams are identical except for some data at the start of the first stream.

To avoid the segment boundary-shifting problem, our second segmentation algorithm, *variable-size segmentation*, uses the same trick used at the chunking level to avoid the boundary-shifting problem: we base the boundaries on landmarks in the content, not distance. Variable-size segmentation operates at the level of chunks (really chunk hashes) rather than bytes and places segment boundaries only at existing chunk boundaries. The start of a chunk is considered to represent a landmark if that chunk’s hash modulo a predetermined divisor is equal to -1. The frequency of landmarks—and hence average segment size—can be controlled by varying the size of the divisor.

To reduce segment-size variation, variable-size segmentation uses TTTD applied to chunks instead of data bytes. The algorithm is the same, except that we move one chunk at a time instead of one byte at a time, and that we use the above notion of what a landmark is. Note that this ignores the lengths of the chunks, treating long and short chunks the same. We obtain the needed TTTD parameters (minimum size, maximum size, primary divisor, and secondary divisor) in the usual way from the desired average size. Thus, for example, with variable-size segmentation, mean size 10 MB segments using 4 KB chunks have from 1,160 to 7,062 chunks with an average of 2,560 chunks, each chunk of which, on average, contains 4 KB of data.

3.2 Choosing champions

Looking up the hooks of an incoming segment S in the sparse index results in a possible set of manifests against which that segment can be deduplicated. However, we

do not necessarily want to use all of those manifests to deduplicate against, since loading manifests from disk is costly. In fact, as we show in Section 4.3, only a few well chosen manifests suffice. So, from among all the manifests produced by querying the sparse index, we choose a few to deduplicate against. We call the chosen manifests champions.

The algorithm by which we choose champions is as follows: we choose champions one at a time until the maximum allowable number of champions are found, or we run out of candidate manifests. Each time we choose, we choose the manifest with the highest non-zero score, where a manifest gets one point for each hook it has in common with S that is not already present in any previously chosen champion. If there is a tie, we choose the manifest most recently stored. The choice of which manifests to choose as champions is done based solely on the hooks in the sparse index; that is, it does not involve any disk accesses.

We don’t give points for hooks belonging to already chosen manifests because those chunks (and the chunks around them by chunk locality) are most likely already covered by the previous champions. Consider the following highly-simplified example showing the hooks of S and three candidate manifests (m_1 – m_3):

S	b	c	d	e	m	n
m_1	a	b	c	d	e	f
m_2	z	a	b	c	d	f
m_3	m	n	o	p	q	r

The manifests are shown in descending order of how many hooks they have in common with S (common hooks shown in bold). Our algorithm chooses m_1 then m_3 , which together cover all the hooks of S , unlike m_1 and m_2 .

3.3 Deduplicating against the champions

Once we have determined the champions for the incoming segment, we load their manifests from disk. A small cache of recently loaded manifests can speed this process up somewhat because adjacent segments sometimes have champions in common.

The hashes of the chunks in the incoming segment are then compared with the hashes in the champions’ manifests in order to find duplicate chunks. We use the SHA1 hash algorithm [15] to make false positives here extremely unlikely. Those chunks that are found not to be present in any of the champions are stored on disk in chunk containers, and a new manifest is created for the incoming segment. The new manifest contains the loca-

tion on disk where each incoming chunk is stored. In the case of chunks that are duplicates of a chunk in one or more of the champions, the location is the location of the existing chunk, which is obtained from the relevant manifest. In the case of new chunks, the on-disk location is where that chunk has just been stored. Once the new manifest is created, it is stored on disk in the manifest store.

Finally, we add entries for this manifest to the sparse index with the manifest's hooks as keys. Some of the hooks may already exist in the sparse index, in which case we add the manifest to the list of manifests that are pointed to by that hook. To conserve space, it may be desirable to set a maximum limit for the number of manifests that can be pointed to by any one hook. If the maximum is reached, the oldest manifest is removed from the list before the newest one is added.

3.4 Avoiding the chunk-lookup disk bottleneck

Notice that there is no full chunk index in our approach, either in RAM or on disk. The only index we maintain in RAM, the sparse index, is much smaller than a full chunk index: for example, if we only sample one out of every 128 hashes, then the sparse index can be 128 times smaller than a full chunk index.

We do have to make a handful of random disk accesses per segment in order to load in champion manifests, but the cost of those seeks is amortized over the thousands of chunks in each segment, leading to acceptable throughput. Thus, we avoid the chunk-lookup disk bottleneck.

3.5 Storing chunks

We do not have room in this paper, alas, to describe how best to store chunks in chunk containers. The scheme described in Zhu *et al.* [28], however, is a pretty good starting point and can be used with our approach. They maintain an open chunk container for each incoming stream, appending each new (unique) chunk to the open container corresponding to the stream it is part of. When a chunk container fills up (they use a fixed size for efficient packing), a new one is opened up.

This process uses chunk locality to group together chunks likely to be retrieved together so that restoration performance is reasonable. Supporting deletion of segments requires additional machinery for merging mostly empty containers, garbage collection (when is it safe to stop storing a shared chunk?), and possibly defragmentation.

3.6 Using less bandwidth

We have described a system where all the raw backup data is fed across the network to the backup system and only then deduplicated, which may consume a lot of network bandwidth. It is possible to use substantially less bandwidth at the cost of some client-side processing if the legacy backup clients could be modified or replaced. One way of doing this is to have the backup client perform the chunking, hashing, and segmentation locally. The client initially sends only a segment's chunks' hashes to the back-end, which performs champion choosing, loads the champion manifests, and then determines which of those chunks need to be stored. The back-end notifies the client of this and the client sends only the chunks that need to be stored, possibly compressed.

4 Experimental Results

In order to test our approach, we built a simulator that allows us to experiment with a number of important parameters, including some parameter values that are infeasible in practice (e.g., using a full index). We apply our simulator to two realistic data sets and report below on locality, overall deduplication, RAM usage, and throughput. We also report briefly on some optimizations and an ongoing productization that validates our approach.

4.1 Simulator

Our simulator takes as input a series of (chunk hash, length) pairs, divides it into segments, determines the champions for each segment, and then calculates the amount of deduplication obtained. Available knobs include type of segmentation (fixed or variable size), mean segment size, sampling rate, maximum number of champions loadable per segment, how many manifest IDs to keep per hook in the sparse index, and whether or not to use a simple form of manifest caching (see Section 4.7).

We (or others when privacy is a concern) run a small tool we have written called *chunklite* in order to produce chunk information for the simulator. Chunklite reads from either a mounted tape or a list of files, chunking using the TTTD chunking algorithm [13]. Except where we say otherwise, all experiments use chunklite's default 4 KB mean chunk size,² which we find a good trade-off between maximizing deduplication and minimizing per-chunk overhead.

The simulator produces various statistics, including the sum of lengths of every input chunk (original size) and the sum of lengths of every non-removed chunk (deduplicated size). The estimated deduplication factor is then original size/deduplicated size.

4.2 Data sets

We report results for two data sets. The first data set, which we call *Workgroup*, is composed of a semi-regular series of backups of the desktop PCs of a group of 20 engineers taken over a period of three months. Although the original collection included only an initial full and later weekday incrementals for each machine, we have generated synthetic fulls at the end of each week for which incrementals are available by applying that week's incrementals to the last full. The synthetic fulls replace the last incremental for their week; in this way, we simulate a more typical daily incremental and weekly full backup schedule. We are unable to simulate file deletions because this information is missing from the collection.

Altogether, there are 154 fulls and 392 incrementals in this 3.8 TB data set, which consists of each of these backup snapshots tar'ed up without compression in the order they were taken. We believe this data set is representative of a small corporate workgroup being backed up via tar directly to a NAS interface. Note that because these machines are only powered up during workdays and because the synthetic fulls replace the last day of the week's back up, the ratio of incrementals to fulls (2.5) is lower than would be the case for a server (6 or 7).

The second data set, which we call *SMB*, is intended, by contrast, to be representative of a small or medium business server backed up to virtual tape. It contains two weeks (3 fulls, 12 incrementals) of Oracle & Exchange data backed up via Symantec's NetBackup to virtual tape. The Exchange data was synthetic data generated by the Microsoft Exchange Server 2003 Load Simulator (LoadSim) tool [19], while the Oracle data was created by inserting rows from a real 1+ TB Oracle database belonging to a compliance test group combined with a small number of random deletes and updates. This data set occupies 0.6 TB and has less duplication than one might expect because Exchange already uses single instance storage (each message is stored only once no matter how many users receive it) and because NetBackup does true Exchange incrementals, saving only new/changed messages.

We have chosen data sets with daily incrementals and weekly fulls rather than just daily fulls because such data sets are harder to deduplicate well, and thus provide a better test of any deduplication system. Incrementals are harder to deduplicate because they contain less duplicate material and because they have less locality: any given incremental segment likely contains files from many segments of the previous full whereas a full segment may only contain files from one or two segments of the previous full. Series of all fulls do generate higher deduplication factors, beloved of marketing departments everywhere, however.

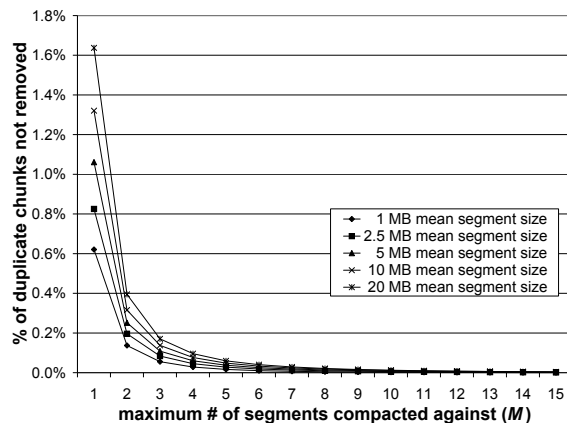


Figure 2: **Conservative estimate (GREEDY) of deduplication effectiveness obtainable by deduplicating each segment against up to M prior segments** for data set *Workgroup*. Shown are results for 5 different segment sizes, with all segments chosen via variable-size segmentation.

4.3 Locality

In order for our approach to work, there must be sufficient locality present in real backup streams. In particular, we need locality at the scale of our segment size so that most of the deduplication possible for a given segment can be obtained by deduplicating it against a small number of prior segments. The existence of such locality is a necessary, but not sufficient condition: the existence of such segments does not automatically imply that sparse indexing or any other method can efficiently find them.

Whether or not such locality exists is an empirical question, which we find to be overwhelmingly answered in the affirmative. Figures 2 and 3 show a conservative estimate of this locality for our data sets for a variety of segment sizes. Here, we show how well segment-based deduplication could work given near-perfect segment choice when each segment of the given data set can only be deduplicated against a small number M of prior segments. We measure deduplication effectiveness by the percentage of duplicate chunks that deduplication fails to remove; the smaller this number, the better the deduplication.

Because computing the optimal segments to deduplicate against is infeasible in practice, we instead estimate the deduplication effectiveness possible by using a simple greedy algorithm (GREEDY) that chooses the segments to deduplicate a given segment S against one at a time, each time choosing the segment that will produce the maximum additional deduplication. While GREEDY

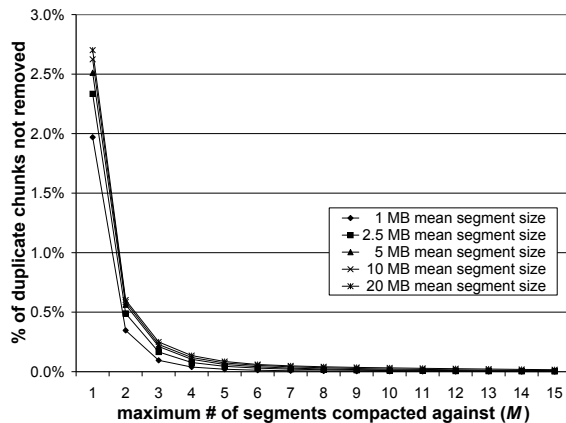


Figure 3: **Conservative estimate (GREEDY) of deduplication effectiveness obtainable by deduplicating each segment against up to M prior segments** for data set **SMB**. Shown are results for 5 different segment sizes, with all segments chosen via variable-size segmentation.

does an excellent job of choosing segments, it consumes too much RAM to ever be practical.

As you can see, there is a great deal of locality at these scales: deduplicating each input segment against only 2 prior segments can suffice to remove all but 1% of the duplicate chunks (0.1% requires only 3 more). Not shown is the zero segment case ($M=0$) where 93-98% of duplicate chunks remain due to duplication within segments (segments are automatically deduplicated against themselves). Larger segment sizes yield slightly less locality, presumably because larger pieces of incrementals include data from more faraway places.

Likely sources of locality in backup streams include writing out entire large items even when only a small part has changed (e.g., Microsoft Outlook's mostly append-only PST files, which are often hundreds of megabytes long), locality in the order items are scanned (e.g., always scanning files in alphabetical order), and the tendency for changes to be clustered in small areas.

4.4 Overall deduplication

How much of this locality are we able to exploit using sampling? Figure 4 addresses this point by showing for data set **Workgroup** and 10 MB variable size segments how much of the possible deduplication efficiency we obtain. Even with a sampling rate as low as 1/128, we remove all but 1.4% of the duplicate data given a maximum of 10 or more champions per segment (0.7% for 1/64).

Figures 5 and 6 show the overall deduplication produced by applying our approach to the two data sets.

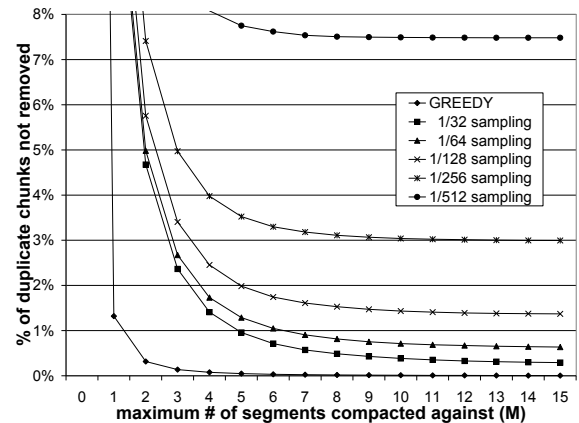


Figure 4: **Deduplication efficiency obtained by using sparse indexing with 10 MB average-sized segments for various maximum numbers of champions (M) and sampling rates** for data set **Workgroup**. Shown for comparison is GREEDY's results given the same data. Variable-size segmentation was used.

As can be seen, the degree of deduplication achieved falls off as the sampling rate decreases and as the segment size decreases. The amount of deduplication remains roughly constant as sampling rate is traded off against segment size: halving the sampling rate and doubling the mean segment size leaves deduplication roughly the same. This can be seen most easily in Figure 7, which plots overall deduplication versus the average number of hooks per segment (equal to segment size/chunk size \times sampling rate). We believe this relationship reflects the fact that deduplication quality using sparse indexing depends foremost on the number of hooks per segment.

Note that these figures show simulated deduplication, not real deduplication. In particular, they take into account only the space required to keep the data of the non-deduplicated chunks. Including container padding, the space required to store manifests, and other overhead would reduce these numbers somewhat. Similar overhead exists in all backup systems that use chunk-based deduplication. On the other hand, these numbers do not include any form of local compression of chunk data. In practice, chunks would be compressed (e.g., by Ziv-Lempel [29]), either individually or in groups, before storing to disk. Such compression usually adds an additional factor of 1.5–2.5.

Using variable instead of fixed-size segmentation improves deduplication using sparse indexing as can be seen from Figure 8. This improvement is due to increased locality: with fixed-size segmentation, there are more segments that produce substantial deduplication

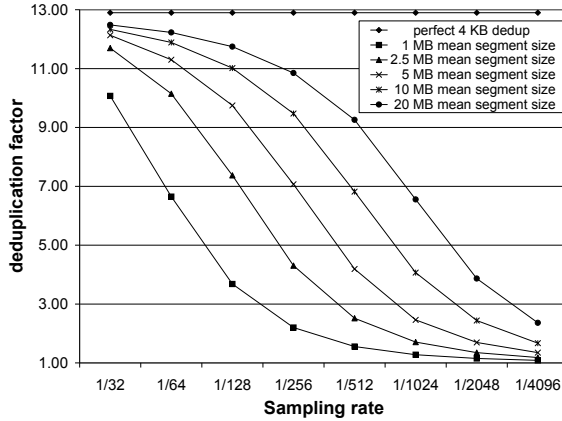


Figure 5: Deduplication produced using sparse indexing with up to 10 champions ($M=10$) for various sampling rates and segment sizes for data set **Workgroup**. For each point, the deduplication factor (deduplicated size/original size) is shown. Shown for comparison is perfect 4 KB deduplication, wherein all duplicate chunks are removed. Variable-size segmentation was used.

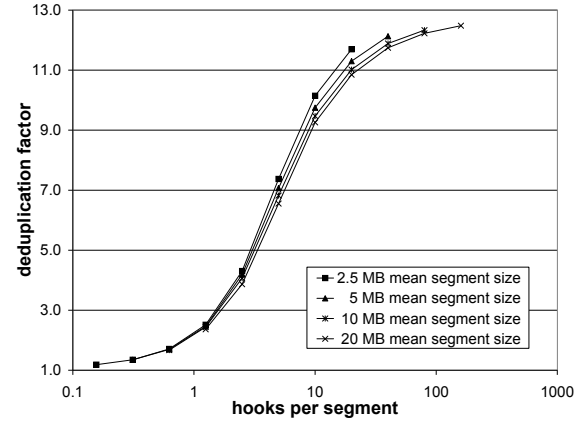


Figure 7: Deduplication produced using sparse indexing with up to 10 champions ($M=10$) versus the average number of hooks per segment for various sampling rates and segment sizes for data set **Workgroup**. For each segment size, sampling rates (from right to left) of 1/32, 1/64, 1/128, 1/256, 1/512, 1/1024, 1/2048, and 1/4096 are shown. Variable-size segmentation was used.

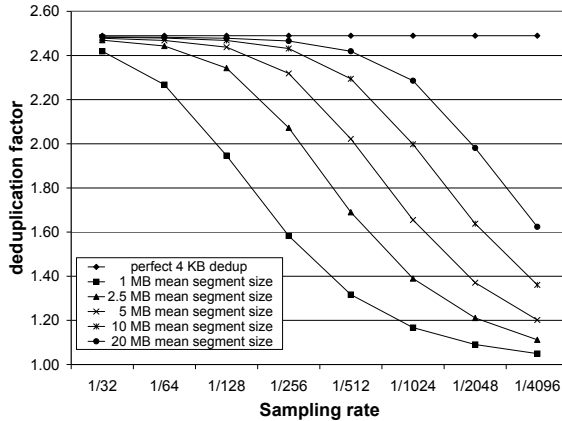


Figure 6: Deduplication produced using sparse indexing with up to 10 champions ($M=10$) for various sampling rates and segment sizes for data set **SMB**. For each point, the deduplication factor (deduplicated size/original size) is shown. Shown for comparison is perfect 4 KB deduplication, wherein all duplicate chunks are removed. Variable-size segmentation was used.

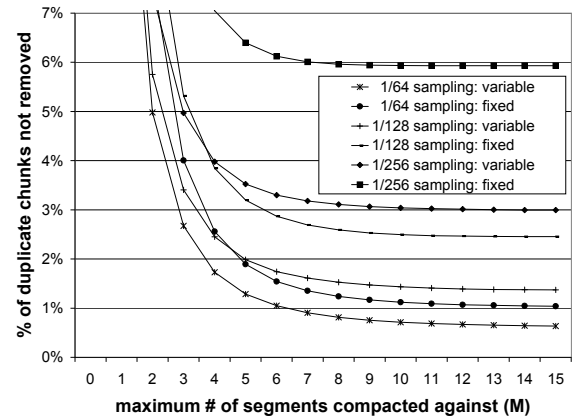


Figure 8: Fixed versus variable-size segmentation with 10 MB average size segments for selected sampling rates for data set **Workgroup**.

that must be found in order to achieve high levels of deduplication quality. Because this introduces more opportunities for serious mistakes (e.g., missing such a segment due to poor sampling), sparse indexing does substantially worse with fixed-size segmentation.

4.5 RAM usage and comparison with Zhu *et al.*

Since one of the main objectives of this paper is to argue that our approach significantly reduces RAM usage for comparable deduplication and throughput to existing approaches, we briefly describe here the approach used by Zhu *et al.* [28], which we call the Bloom Filter with Paged Full Index (BFPFI) approach.

BFPFI uses a full disk-based index of every chunk hash. To avoid having to access the disk for every hash lookup, it employs a Bloom Filter and a cache of chunk container indexes. The Bloom Filter uses one byte of RAM per hash and contains the hash of every chunk in the store. If the Bloom filter does not indicate that an incoming chunk is already in the store, then there is no need to consult the chunk index. Otherwise, the cache is searched and only if it fails to contain the given chunk's hash, is the on-disk full chunk index consulted. Each time the on-disk index must be consulted, the index of the chunk container that contains the given chunk (if any) is paged into memory.

The hit rate of the BFPFI cache (and hence the overall throughput) depends on the degree of chunk locality of the input data: because chunk containers contain chunks that occurred together before, high chunk locality implies a high hit rate. The only parameter that impacts the deduplication factor in BFPFI is the average chunk size, since it finds all the duplicate chunks. Smaller chunk sizes increase the deduplication factor at the cost of requiring more RAM for the Bloom filter.

Both approaches degrade under conditions of poor chunk locality: with BFPFI, throughput degrades, whereas with sparse indexing, deduplication quality degrades. Unlike with BFPFI, with sparse indexing it is possible to guarantee a minimum throughput by imposing a maximum number of champions, which can be important given today's restricted backup window times. It is, of course, impossible to guarantee a minimum deduplication factor because the maximum deduplication possible is limited by characteristics of the input data that are beyond the control of any store.

The amount of RAM required by one of our sparse indexes or the Bloom filter of the BFPFI approach is linearly proportional to the maximum possible number of unique chunks in that store. Accordingly, we plot RAM usage as the ratio of RAM required per amount of physical disk storage. Figure 9 shows the estimated

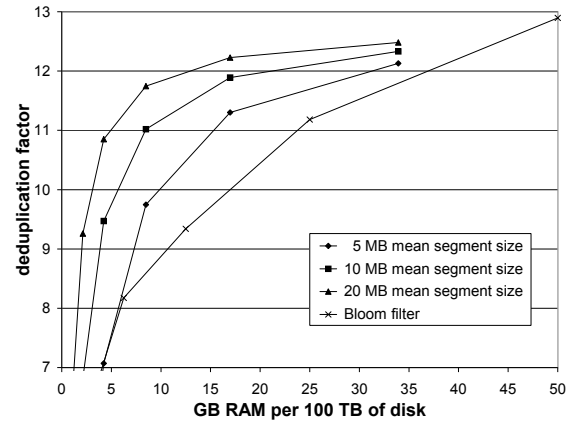


Figure 9: **RAM space required per 100 TB of disk for sparse indexing with up to 10 champions ($M=10$) and for a Bloom filter.** For each point, the deduplication factor for data set Workgroup is shown. Each sparse indexing series shows points for sampling rates of (right to left) $1/32$, $1/64$, $1/128$, $1/256$, and $1/512$ while the Bloom filter series shows points for chunk sizes of 4, 8, 16, and 32 KB. Variable-size segmentation was used.

RAM usage of sparse indexes with 5, 10, and 20 MB variable-sized segments as well as the Bloom filter used by BFPFI. We assume here a local compression factor of 2, which allows 100 TB of disk to store twice as many chunks as would otherwise be possible. Because it is easy to achieve good RAM usage if deduplication quality can be neglected, we also show the deduplication factor for data set Workgroup for each case.

You will note that our approach uses substantially less RAM than BFPFI for the same quality of deduplication. For example, for a store with 100 TB of disk, a sparse index with 10 MB segments and $1/64$ sampling requires 17 GB whereas we estimate a Bloom filter would require 36 GB for an equivalent level of deduplication. Alternatively, starting with a Bloom filter using 8 KB chunks (the value used by Zhu *et al.* [28]), which requires 25 GB for 100 TB, we estimate we can get the same deduplication quality (using 4 KB chunks) but use only 10 GB (10 MB segments) or 6 GB (20 MB segments) of RAM. For comparison, the Jumbo Store [14], which keeps a full chunk index in RAM, would need 1,500 GB for the second case.

A sparse index has one key for each unique hook encountered; when using a sampling rate of $1/s$, on average $1/s$ of unique chunks will have a hash which qualifies as a hook. Because of the random nature of hashes and the law of large numbers (we are dealing with billions of unique chunks), we can treat this average as a maximum for estimation purposes. To conserve RAM needed

by our simulated sparse indexes, we generally limit the number of manifest IDs per hook in our sparse indexing experiments to 1; that is, for each hook, we simulate keeping the ID of only the last manifest containing that hook. This slightly decreases deduplication quality (see Section 4.7), but saves a lot of RAM.

Such a sparse index needs to be big enough to hold u/s keys, each of which has exactly 1 entry, where u is the maximum number of unique chunks possible. The sampling rate is thus the primary factor controlling RAM usage for our experiments. The actual space is ku/s where k is a constant depending on the exact data structure implementation used. For this figure, we have used $k = 21.7$ bytes based on using a chained hash table with a maximum 70% load factor, 4-byte internal pointers, 8-byte manifest IDs, and 4 key check bytes per entry. Using only a few bytes of the key saves substantial RAM but means the index can—very rarely—make mistakes; this may occasionally result in the wrong champion being selected, but is unlikely to substantially alter the overall deduplication quality. We calculate the size of the BFPFI Bloom filter per Zhu *et al.* as 1 byte per unique chunk [28].

Additional RAM is needed for both approaches for per stream buffers. In our case, the per stream space is proportional to the segment size.

4.6 Throughput

Because we do not move around or even simulate moving around chunk data, we cannot estimate overall read or write throughput. However, because we do collect statistics on how many champions are loaded per segment, we can estimate the I/O burden that loading champions places on a system using our approach. Aside from this I/O and writing out new manifests, the only other I/O our system needs to do when ingesting data is that required by any other deduplicating store: reading in the input data and writing out the non-deduplicated chunks.

Similarly, the majority of the computation required by our approach—chunking, hashing, and compression—also must be done by any chunk-based deduplication engine. For an alternative way of getting a handle on the throughput our approach can support, see Section 4.8 where we briefly describe some early throughput measurements of a product embodying our approach.

Figure 10 shows the average number of champion manifests actually loaded per segment for the data set Workgroup with up to 10 champions per segment allowed. The equivalent chart for SMB (not shown) is similar, but scaled down by a factor of 2/3. You will notice that the average number loaded is substantially less than the maximum allowed, 10. This is primarily because most segments in these data sets can

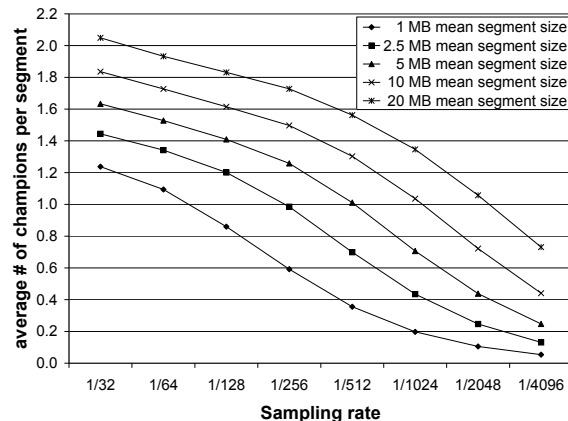


Figure 10: Average number of champions actually loaded per segment using sparse indexing with up to 10 champions ($M=10$) for various sampling rates and segment sizes for data set Workgroup. Variable-size segmentation was used.

be completely deduplicated using only a few champions (GREEDY does not load substantially more champions than 1/32-sampling). Lower sampling rates result in even fewer champions being loaded because sparser indexes result in fewer candidate champions being identified.

Loading a champion manifest requires a random seek followed by a quite small amount of sequential reading (manifests are a hundredth of the size of segments and measured in KBs). Accordingly, the I/O burden due to loading champions is best measured in terms of the average number of seeks (equivalently champions loaded) per unit of input data. Figure 11 shows this information for the Workgroup data set. Note that the ordering of segment sizes has reversed: although bigger segments load more champions each, they load their champions so much less frequently that their overall rate of loading champions per megabyte of input data, and hence, their I/O burden is less.

If we conservatively assume that loading a champion manifest takes 20 ms and that we load 0.2 champions per megabyte on average, then a single drive doing nothing else could support a rate of $1/(0.2 \cdot 20 \text{ ms/MB}) = 250 \text{ MB/s}$. Of course, as we mentioned above, there is other I/O that needs to be done as well. However, in practice deduplication systems are usually deployed with 10 or more drives so the real number before other I/O needs is more like 2.5 GB/s. This is a sufficiently light burden that we expect that some other component of the system will be the throughput bottleneck.

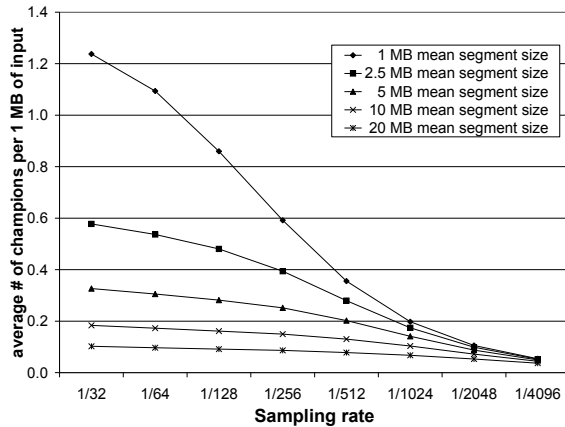


Figure 11: Average number of champions actually loaded per 1 MB of input data using sparse indexing with up to 10 champions ($M=10$) for various sampling rates and segment sizes for data set **Workgroup**. Variable-size segmentation was used.

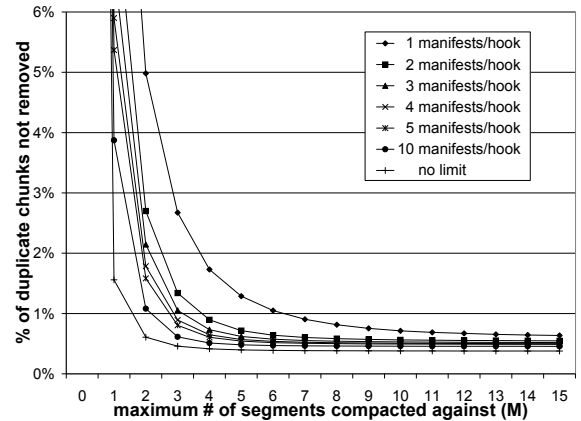


Figure 12: Deduplication efficiency obtained as the maximum number of manifest IDs kept per hook varies for 10 MB average size segments and a sampling rate of 1/64 for data set **Workgroup**. Variable-size segmentation was used.

4.7 Optimization

Figure 12 shows how capping the number of manifest IDs kept per hook in the sparse index affects deduplication quality. Keeping only one manifest ID per hook does reduce deduplication somewhat (99.29% versus 99.44% duplicate chunks removed for $M = 10$ here), but greatly reduces the amount of RAM required for the sparse index.

All the experiments we have reported on do not use any manifest caching at all. While the design of our simulator unfortunately makes it hard to implement manifest caching correctly (we compute the i th champion for each segment in parallel), we were able to conservatively approximate a cache just large enough to hold the champions from the previous segment.³ We find that even with this suboptimal implementation, manifest caching reduces champions *loaded* per segment and slightly improves deduplication quality. For 10 MB variable size segments, $M = 10$, and 1 in 64 sampling for data set **Workgroup**, for example, our version of manifest caching lowers the average number of champions loaded per segment by 3.9% and improves the deduplication factor by 1.1%.

4.8 Productization

Our approach is being used to build a family of VTL products that use deduplication internally to increase the amount of data they can support. Already on the market are the HP D2D2500 and the D2D4000. Most of the work described in this paper, however, was done before

even a prototype of these products was available.

A third-party testing firm, Binary Testing Ltd., was hired by HP to test the D2D4000's deduplication performance [5]. The D2D4000 configuration they tested has 6 750 GB disk drives running RAID 6, 8 GB RAM, 2 AMD Opteron 3 Ghz dual core processors, and a 4 Gb fiber channel link. We report a few representative excerpts from their report here: changing 0.4% of every file of a 4 GB file server data set every day and taking fulls for three months produced a deduplication factor of 69.2; the same change schedule applied to a 4 GB exchange server produced a factor of 24.9. Instead changing only 20% of the items every day but by 5% yielded factors of 25.5 (for the file server) and 40.3 (for the Exchange server). Note that these numbers include all overhead and local compression.

Preliminary throughput testing of a similar system with 12 750 GB disk drives shows write rates of 90 MB/s (1 stream) and 120 MB/s (4 streams) and read rates of 40-50 MB/s (1 stream) and 25-35 MB/s (4 streams). The restore path was still being optimized when these measurements were taken, so those numbers may improve substantially. We believe these product results validate our approach, and demonstrate that we have not overlooked any crucial points.

5 Related Work

Chunking has been used to weed out near duplicates in repositories [16], conserve network bandwidth [20], and reduce storage space requirements [1, 22, 23, 27]. It has also been used to synchronize large data sets reliably

while conserving network bandwidth [14, 17].

Archival and backup storage systems detect duplicate data at granularities that range from an entire file, as in EMC's Centera [12], down to individual fixed-size disk blocks, as in Venti [22], and variable-size data chunks, as in the Low-Bandwidth Network File System [20]. Variable-sized chunking has also been used in the commercial sector, for example, by Data Domain and Riverbed Technology. Deep Store [27] is a large-scale archival storage system that uses both delta compression [2, 11] and chunking to reduce storage space requirements. How much deduplication is obtained depends on the inherent content overlaps in the data, the granularity of chunks, and the chunking method [21]. Deduplication using chunking can be quite effective for data that evolves slowly (mainly) through small changes, additions, and deletions [26].

Chunking is just one of the methods in the literature used to detect similarities or content overlap between documents. Shingling [8] was developed by Broder for near duplicate detection in web pages. Manber [18], Brin *et al.* [7], and Forman *et al.* [16] have also developed techniques for finding similarities between documents in large repositories.

Various approaches have been used to reduce disk accesses when querying an index. Database buffer management strategies [10] that aim to efficiently maintain a 'working set' of rows of the index in a buffer cache have been well researched. However, these strategies do not work in the case of chunk-based deduplication because chunk IDs are random hashes for which it is not possible to identify or maintain a working set.

Bloom filters [6] have also been used to minimize index accesses. A Bloom filter, which can give false positives but not false negatives, can be used to determine the existence of a key in an index before actually querying the index. If the Bloom filter does not contain the key, then the index does not need to be queried thereby eliminating both an index and possibly a disk access. Bloom filters have been used by large scale distributed storage systems such as Google's BigTable [9] and by Data Domain [28].

Besides using Bloom filters to improve the deduplication throughput, Data Domain exploits chunk locality for index caching as well as for laying out chunks on disk. By using these techniques Data Domain can avoid a large number of disk accesses related to index queries. Venti uses a disk-based hash table divided into buckets where a hash function is used to map chunk hashes to appropriate buckets. To improve the index lookup performance, Venti uses caching, striping, and write buffering. Foundation [23] is an archival storage system that preserves users' data and dependencies by capturing and storing regular snapshots of every users'

virtual machine. Chunking is used to deduplicate the snapshots. Foundation also uses a combination of Bloom filters and locality-friendly on-disk layouts to improve the performance of index lookups.

6 Conclusions

D2D backup is increasingly becoming the backup solution of choice, and deduplication is an essential feature of D2D backup. Our experimental evaluation has shown that there exists a lot of locality within backup data at the small number of megabytes scale. Our approach exploits this locality to solve the chunk-lookup disk bottleneck problem. Through content-based segmentation, sampling, and sparse indexing, we divide incoming streams into segments, identify similar existing segments, and deduplicate against them, yielding excellent deduplication and throughput while requiring little RAM.

While our approach allows a few duplicate chunks to be stored, we more than make up for this loss of deduplication by using a smaller chunk size (possible because of the small RAM requirements), which produces greater deduplication. Compared with the BFPFI approach, we use less than half the RAM for an equivalent high level of deduplication. The practicality of our approach has been demonstrated by its being used as the basis of a HP product family.

Acknowledgments

We would like to thank Graham Perry, John Czerkowiec, David Falkinder, and Kevin Collins for their help and support. We would also like to thank the anonymous FAST'09 reviewers and Greg Ganger, our shepherd. This work was done while the third author was an intern at HP.

References

- [1] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, December 2002), pp. 1–14.
- [2] AJTAI, M., BURNS, R., FAGIN, R., LONG, D. D. E., AND STOCKMEYER, L. Compactly encoding unstructured inputs with differential compression. *Journal of the Association for Computing Machinery* 49, 3 (May 2002), 318–367.
- [3] ASARO, T., AND BIGGAR, H. Data De-duplication and Disk-to-Disk Backup Systems: Technical and Business Considerations. *The Enterprise Strategy Group* (July 2007).
- [4] BIGGAR, H. Experiencing Data De-Duplication: Improving Efficiency and Reducing Capacity Requirements. *The Enterprise Strategy Group* (Feb. 2007).

- [5] BINARY TESTING LTD. HP StorageWorks D2D4000 Backup System: a report and full performance test on Hewlett-Packard's SME data deduplication appliance. Available at http://h18006.www1.hp.com/products/storageworks/d2d_4000/relatedinfo.html, July 2008.
- [6] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13, 7 (1970), 422–426.
- [7] BRIN, S., DAVIS, J., AND GARCÍA-MOLINA, H. Copy detection mechanisms for digital documents. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, United States, 1995), ACM Press, pp. 398–409.
- [8] BRODER, A. Z. On the resemblance and containment of documents. In *SEQUENCES '97: Proceedings of the Compression and Complexity of Sequences 1997* (Washington, DC, USA, 1997), IEEE Computer Society, pp. 21–29.
- [9] CHANG, F., DEAN, J., GHAMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In *OSDI'06: Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation* (Seattle, WA, 2006), pp. 205–218.
- [10] CHOU, H.-T., AND DEWITT, D. J. An evaluation of buffer management strategies for relational database systems. In *Proc. of the 11th Conference on Very Large Databases (VLDB)* (Stockholm, Sweden, 1985), VLDB Endowment, pp. 127–141.
- [11] DOUGLIS, F., AND IYENGAR, A. Application-specific delta-encoding via resemblance detection. In *Proceedings of the 2003 USENIX Annual Technical Conference* (San Antonio, Texas, June 2003), pp. 113–126.
- [12] EMC CORPORATION. EMC Centera: Content Addressed Storage System, Data Sheet, April 2002.
- [13] ESHGHI, K. A framework for analyzing and improving content-based chunking algorithms. Tech. Rep. HPL-2005-30(R.1), Hewlett Packard Laboratories, Palo Alto, 2005.
- [14] ESHGHI, K., LILLIBRIDGE, M., WILCOCK, L., BELROSE, G., AND HAWKES, R. Jumbo Store: Providing efficient incremental upload and versioning for a utility rendering service. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)* (San Jose, CA, February 2007), USENIX Association, pp. 123–138.
- [15] Federal Information Processing Standard (FIPS) 180–3: Secure Hash Standard (SHS). Tech. Rep. 180–3, National Institute of Standards and Technology (NIST), Gaithersburg, MD, Oct 2008.
- [16] FORMAN, G., ESHGHI, K., AND CHIOCCETTI, S. Finding similar files in large document repositories. In *Proceeding of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)* (Chicago, IL, USA, August 2005), ACM Press, pp. 394–400.
- [17] JAIN, N., DAHLIN, M., AND TEWARI, R. TAPER: Tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)* (San Francisco, CA, USA, December 2005), pp. 281–294.
- [18] MANBER, U. Finding similar files in a large file system. In *Proceedings of the Winter 1994 USENIX Technical Conference* (San Francisco, CA, USA, January 1994), pp. 1–10.
- [19] Microsoft exchange server 2003 load simulator. Download available at <http://www.microsoft.com/downloads>, February 2006.
- [20] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (Banff, Alberta, Canada, October 2001), ACM Press, pp. 174–187.
- [21] POLICRONIADES, C., AND PRATT, I. Alternatives for detecting redundancy in storage systems data. In *Proc. of the General Track, 2004 USENIX Annual Technical Conference* (Boston, MA, USA, June 2004), USENIX Association, pp. 73–86.
- [22] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies* (Monterey, CA, USA, January 2002), USENIX Association, pp. 89–101.
- [23] RHEA, S., COX, R., AND PESTEREV, A. Fast, inexpensive content-addressed storage in Foundation. In *Proceedings of the 2008 USENIX Annual Technical Conference* (Boston, Massachusetts, June 2008), pp. 143–156.
- [24] SHAPIRO, L. D. Join processing in database systems with large main memories. *ACM Transactions on Database Systems* 11, 3 (1986), 239–264.
- [25] TOLIA, N., KOZUCH, M., SATYANARAYANAN, M., KARP, B., BRESSOUD, T., AND PERRIG, A. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the General Track, 2003 USENIX Annual Technical Conference* (San Antonio, Texas, June 2003), USENIX Association, pp. 127–140.
- [26] YOU, L. L., AND KARAMANOLIS, C. Evaluation of efficient archival storage techniques. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies* (College Park, MD, April 2004).
- [27] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep Store: An archival storage system architecture. In *Proc. of the 21st International Conference on Data Engineering (ICDE '05)* (Tokyo, Japan, April 2005), IEEE, pp. 804–815.
- [28] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)* (San Jose, CA, USA, February 2008), USENIX Association, pp. 269–282.
- [29] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343.

Notes

¹ E.g., 30 fulls with 5% data change/day might when deduplicated occupy the space of $1 + 29 \cdot 5\% = 2.45$ fulls so the extra full worth of space needed by out-of-line amounts to requiring 29% more disk space.

² Standard TTTD parameters yield for this chunk size a minimum chunk size of 1,856 and a maximum chunk size of 11,299 using primary divisor 2,179 and secondary divisor 1,099.

³ We get the effect of a size- M manifest cache by causing our simulator to do the following: each time it chooses the i th champion for a given segment, it immediately also deduplicates the given segment against the i th champion of the immediately preceding segment as well.

Generating Realistic *Impressions* for File-System Benchmarking

Nitin Agrawal, Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau
Department of Computer Sciences, University of Wisconsin-Madison
{nitina, dusseau, remzi}@cs.wisc.edu

Abstract

The performance of file systems and related software depends on characteristics of the underlying file-system image (*i.e.*, file-system metadata and file contents). Unfortunately, rather than benchmarking with realistic file-system images, most system designers and evaluators rely on *ad hoc* assumptions and (often inaccurate) rules of thumb. Furthermore, the lack of standardization and reproducibility makes file system benchmarking ineffective. To remedy these problems, we develop *Impressions*, a framework to generate statistically accurate file-system images with realistic metadata and content. *Impressions* is flexible, supporting user-specified constraints on various file-system parameters using a number of statistical techniques to generate consistent images. In this paper we present the design, implementation and evaluation of *Impressions*, and demonstrate its utility using desktop search as a case study. We believe *Impressions* will prove to be useful for system developers and users alike.

1 Introduction

File system benchmarking is in a state of disarray. In spite of tremendous advances in file system design, the approaches for benchmarking still lag far behind. The goal of benchmarking is to understand how the system under evaluation will perform under real-world conditions and how it compares to other systems; however, recreating real-world conditions for the purposes of benchmarking file systems has proven challenging. The two main challenges in achieving this goal are generating representative *workloads*, and creating realistic *file-system state*.

While creating representative workloads is not an entirely solved problem, significant steps have been taken towards this goal. Empirical studies of file-system access patterns [4, 19, 33] and file-system activity traces [38, 45] have led to work on synthetic workload generators [2, 14] and methods for trace replay [3, 26].

The second, and perhaps more difficult challenge, is to recreate the file-system *state* such that it is representative

of the target usage scenario. Several factors contribute to file-system state, important amongst them are the *in-memory* state (contents of the buffer cache), the *on-disk* state (disk layout and fragmentation) and the characteristics of the *file-system image* (files and directories belonging to the namespace and file contents).

One well understood contributor to state is the *in-memory* state of the file system. Previous work has shown that the contents of the cache can have significant impact on the performance results [11]. Therefore, system initialization during benchmarking typically consists of a cache “warm-up” phase wherein the workload is run for some time prior to the actual measurement phase. Another important factor is the *on-disk* state of the file system, or the degree of *fragmentation*; it is a measure of how the disk blocks belonging to the file system are laid out on disk. Previous work has shown that fragmentation can adversely affect performance of a file system [44]. Thus, prior to benchmarking, a file system should undergo *aging* by replaying a workload similar to that experienced by a real file system over a period of time [44].

Surprisingly, one key contributor to file-system state has been largely ignored – the characteristics of the *file-system image*. The properties of file-system metadata and the actual content within the files are key contributors to file-system state, and can have a significant impact on the performance of a system. Properties of file-system metadata includes information on how directories are organized in the file-system namespace, how files are organized into directories, and the distributions for various file attributes such as size, depth, and extension type. Consider a simple example: the time taken for a `find` operation to traverse a file system while searching for a file name depends on a number of attributes of the file-system image, including the depth of the file-system tree and the total number of files. Similarly, the time taken for a `grep` operation to search for a keyword also depends on the type of files (*i.e.*, binary vs. others) and the file content.

File-system benchmarking frequently requires this sort of information on file systems, much of which is

Paper	Description	Used to measure
HAC [17]	File system with 17000 files totaling 150 MB	Time and space needed to create a Glimpse index
IRON [36]	None provided	Checksum and metadata replication overhead; parity block overhead for user files
LBFS [30]	10702 files from /usr/local, total size 354 MB	Performance of LBFS chunking algorithm
LISFS [34]	633 MP3 files, 860 program files, 11502 man pages	Disk space overhead; performance of search-like activities: UNIX find and LISFS lookup
PAST [40]	2 million files, mean size 86 KB, median 4 KB, largest file size 2.7 GB, smallest 0 Bytes, total size 166.6 GB	File insertion, global storage utilization in a P2P system
Pastiche [9]	File system with 1641 files, 109 dirs, 13.4 MB total size	Performance of backup and restore utilities
Pergamum [47]	Randomly generated files of “several” megabytes	Data transfer performance
Samsara [10]	File system with 1676 files and 13 MB total size	Data transfer and querying performance, load during querying
Segank [46]	5-deep directory tree, 5 subdirs and 10 8 KB files per directory	Performance of Segank: volume update, creation of read-only snapshot, read from new snapshot
SFS read-only [15]	1000 files distributed evenly across 10 directories and contain random data	Single client/single server read performance
TFS [7]	Files taken from /usr to get “realistic” mix of file sizes	Performance with varying contribution of space from local file systems
WAFL backup [20]	188 GB and 129 GB volumes taken from the Engineering department	Performance of physical and logical backup, and recovery strategies
yFS [49]	Avg. file size 16 KB, avg. number of files per directory 64, random file names	Performance under various benchmarks (file creation, deletion)

Table 1: Choice of file system parameters in prior research.

available in the form of empirical studies of file-system contents [1, 12, 21, 29, 41, 42]. These studies focus on measuring and modeling different aspects of file-system metadata by collecting snapshots of file-system images from real machines. The studies range from a few machines to tens of thousands of machines across different operating systems and usage environments. Collecting and analyzing this data provides useful information on how file systems are used in real operating conditions.

In spite of the wealth of information available in file-system studies, system designers and evaluators continue to rely on *ad hoc* assumptions and often inaccurate rules of thumb. Table 1 presents evidence to confirm this hypothesis; it contains a (partial) list of publications from top-tier systems conferences in the last ten years that required a test file-system image for evaluation. We present both the description of the file-system image provided in the paper and the intended goal of the evaluation.

In the table, there are several examples where a new file system or application design is evaluated on the evaluator’s personal file system without describing its properties in sufficient detail for it to be reproduced [7, 20, 36]. In others, the description is limited to coarse-grained measures such as the total file-system size and the number of files, even though other file-system attributes (*e.g.*, tree depth) are relevant to measuring performance or storage space overheads [9, 10, 17, 30]. File systems are also sometimes generated with parameters chosen randomly [47, 49], or chosen without explanation of the significance of the values [15, 34, 46]. Occasionally, the

parameters are specified in greater detail [40], but not enough to recreate the original file system.

The important lesson to be learnt here is that there is no standard technique to systematically include information on file-system images for experimentation. For this reason, we find that more often than not, the choices made are arbitrary, suited for ease-of-use more than accuracy and completeness. Furthermore, the lack of standardization and reproducibility of these choices makes it near-impossible to compare results with other systems.

To address these problems and improve one important aspect of file system benchmarking, we develop *Impressions*, a framework to generate representative and statistically accurate file-system images. Impressions gives the user flexibility to specify one or more parameters from a detailed list of file system parameters (file-system size, number of files, distribution of file sizes, etc.). Impressions incorporates statistical techniques (automatic curve-fitting, resolving multiple constraints, interpolation and extrapolation, etc.) and uses statistical tests for goodness-of-fit to ensure the accuracy of the image.

We believe Impressions will be of great use to system designers, evaluators, and users alike. A casual user looking to create a representative file-system image without worrying about carefully selecting parameters can simply run Impressions with its default settings; Impressions will use pre-specified distributions from file-system studies to create a representative image. A more sophisticated user has the power to individually control the knobs for a comprehensive set of file-system parameters; Im-

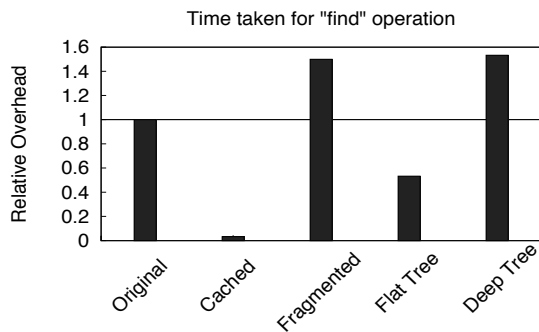


Figure 1: **Impact of directory tree structure.** Shows impact of tree depth on time taken by find. The file systems are created by Impressions using default distributions (Table 2). To exclude effects of the on-disk layout, we ensure a perfect disk layout (layout score 1.0) for all cases except the one with fragmentation (layout score 0.95). The flat tree contains all 100 directories at depth 1; the deep tree has directories successively nested to create a tree of depth 100.

pressions will carefully work out the statistical details to produce a consistent and accurate image. In both cases, Impressions ensures complete reproducibility of the image, by reporting the used distributions, parameter values, and seeds for random number generators.

In this paper we present the design, implementation and evaluation of the Impressions framework (§3), which we intend to release for public use in the near future. Impressions is built with the following design goals:

- *Accuracy*: in generating various statistical constructs to ensure a high degree of statistical rigor.
- *Flexibility*: in allowing users to specify a number of file-system distributions and constraints on parameter values, or in choosing default values.
- *Representativeness*: by incorporating known distributions from file-system studies.
- *Ease of use*: by providing a simple, yet powerful, command-line interface.

Using desktop search as a case study, we demonstrate the usefulness and ease of use of Impressions in quantifying application performance, and in finding application policies and bugs (§4). To bring the paper to a close, we discuss related work (§5), and finally conclude (§6).

2 Extended Motivation

We begin this section by asking a basic question: does file-system structure really matter? We then describe the goals for generating realistic file-system images and discuss existing approaches to do so.

2.1 Does File-System Structure Matter?

Structure and organization of file-system metadata matters for workload performance. Let us take a look at the simple example of a frequently used UNIX utility:

find. Figure 1 shows the relative time taken to run “find /” searching for a file name on a test file system as we vary some parameters of file-system state.

The first bar represents the time taken for the run on the original test file system. Subsequent bars are normalized to this time and show performance for a run with the file-system contents in buffer cache, a fragmented version of the same file system, a file system created by flattening the original directory tree, and finally one by deepening the original directory tree. The graph echoes our understanding of caching and fragmentation, and brings out one aspect that is often overlooked: structure really matters. From this graph we can see that even for a simple workload, the impact of tree depth on performance can be as large as that with fragmentation, and varying tree depths can have significant performance variations (300% between the flat and deep trees in this example).

Assumptions about file-system structure have often trickled into file system design, but no means exist to incorporate the effects of realistic file-system images in a systematic fashion. As a community, we well understand that caching matters, and have begun to pay attention to fragmentation, but when it comes to file-system structure, our approach is surprisingly *laissez faire*.

2.2 Goals for Generating FS Images

We believe that the file-system image used for an evaluation should be *realistic* with respect to the workload; the image should contain a sufficient degree of *detail* to realistically exercise the workload under consideration. An increasing degree of detail will likely require more effort and slow down the process. Thus it is useful to know the degree sufficient for a given evaluation. For example, if the performance of an application simply depends on the size of files in the file system, the chosen file-system image should reflect that. On the other hand, if the performance is also sensitive to the fraction of binary files amongst all files (*e.g.*, to evaluate desktop search indexing), then the file-system image also needs to contain realistic distributions of file extensions.

We walk through some examples that illustrate the different degrees of detail needed in file-system images.

- At one extreme, a system could be completely oblivious to both metadata and content. An example of such a system is a mirroring scheme (RAID-1 [35]) underneath a file system, or a backup utility taking whole-disk backups. The performance of such schemes depends solely on the block traffic.

Alternately, systems could depend on the attributes of the file-system image with different degrees of detail:

- The performance of a system can depend on the amount of file data (number of files and directories, or the size of files and directories, or both) in any

given file system (*e.g.*, a backup utility taking whole file-system snapshots).

- Systems can depend on the structure of the file system namespace and how files are organized in it (*e.g.*, a version control system for a source-code repository).
- Finally, many systems also depend on the actual data stored within the files (*e.g.*, a desktop search engine for a file system, or a spell-checker).

Impressions is designed with this goal of flexibility from the outset. The user is given complete control of a number of file-system parameters, and is provided with an easy to use interface. Transparently, Impressions seamlessly ensures accuracy and representativeness.

2.3 Existing Approaches

One alternate approach to generating realistic file-system images is to randomly select a set of actual images from a corpus, an approach popular in other fields of computer science such as Information Retrieval, Machine Learning and Natural Language Processing [32]. In the case of file systems the corpus would consist of a set of known file-system images (*e.g.*, tarballs). This approach arguably has several limitations which make it difficult and unsuitable for file systems research. First, there are too many parameters required to accurately describe a file-system image that need to be captured in a corpus. Second, without precise control in varying these parameters according to experimental needs, the evaluation can be blind to the actual performance dependencies. Finally, the cost of maintaining and sharing any realistic corpus of file-system images would be prohibitive. The size of the corpus itself would severely restrict its usefulness especially as file systems continue to grow larger.

Unfortunately, these limitations have not deterred researchers from using their personal file systems as a (trivial) substitute for a file-system corpus.

3 The Impressions Framework

In this section we describe the design, implementation and evaluation of Impressions: a framework for generating file-system images with realistic and statistically accurate metadata and content. Impressions is flexible enough to create file-system images with varying configurations, guaranteeing the accuracy of images by incorporating a number of statistical tests and techniques.

We first present a summary of the different modes of operation of Impressions, and then describe the individual statistical constructs in greater detail. Wherever applicable, we evaluate their accuracy and performance.

Parameter	Default Model & Parameters
Directory count w/ depth	Generative model
Directory size (subdirs)	Generative model
File size by count	Lognormal-body ($\alpha_1=0.99994$, $\mu=9.48$, $\sigma=2.46$)
	Pareto-tail ($k=0.91$, $\mathcal{X}_m=512\text{MB}$)
File size by containing bytes	Mixture-of-lognormals ($\alpha_1=0.76$, $\mu_1=14.83$, $\sigma_1=2.35$ $\alpha_2=0.24$, $\mu_2=20.93$, $\sigma_2=1.48$)
Extension popularity	Percentile values
File count w/ depth	Poisson ($\lambda=6.49$)
Bytes with depth	Mean file size values
Directory size (files)	Inverse-polynomial (degree=2, offset=2.36)
File count w/ depth (w/ special directories)	Conditional probabilities (biases for special dirs)
Degree of Fragmentation	Layout score (1.0) or Pre-specified workload

Table 2: **Parameters and default values in Impressions.** List of distributions and their parameter values used in the Default mode.

3.1 Modes of Operation

A system evaluator can use Impressions in different modes of operation, with varying degree of user input.

Sometimes, an evaluator just wants to create a representative file-system image without worrying about the need to carefully select parameters. Hence, in the *automated* mode, Impressions is capable of generating a file-system image with minimal input required from the user (*e.g.*, the size of the desired file-system image), relying on default settings of known empirical distributions to generate representative file-system images. We refer to these distributions as *original* distributions.

At other times, users want more control over the images, for example, to analyze the sensitivity of performance to a given file-system parameter, or to describe a completely different file-system usage scenario. Hence, Impressions supports a *user-specified* mode, where a more sophisticated user has the power to individually control the knobs for a comprehensive set of file-system parameters; we refer to these as user-specified distributions. Impressions carefully works out the statistical details to produce a consistent and accurate image.

In both the cases, Impressions ensures complete reproducibility of the file-system image by reporting the used distributions, their parameter values, and seeds for random number generators.

Impressions can use any dataset or set of parameterized curves for the *original* distributions, leveraging a large body of research on analyzing file-system properties [1, 12, 21, 29, 41, 42]. For illustration, in this paper we use a recent static file-system snapshot dataset made publicly available [1]. The snapshots of file-system metadata were collected over a five-year period representing over 60,000 Windows PC file systems in a large

corporation. These snapshots were used to study distributions and temporal changes in file size, file age, file-type frequency, directory size, namespace structure, file-system population, storage capacity, and degree of file modification. The study also proposed a generative model explaining the creation of file-system namespaces.

Impressions provides a comprehensive set of individually controllable file system parameters. Table 2 lists these parameters along with their default selections. For example, a user may specify the size of the file-system image, the number of files in the file system, and the distribution of file sizes, while selecting default settings for all other distributions. In this case, Impressions will ensure that the resulting file-system image adheres to the default distributions while maintaining the user-specified invariants.

3.2 Basic Techniques

The goal of Impressions is to generate realistic file-system images, giving the user complete flexibility and control to decide the extent of accuracy and detail. To achieve this, Impressions relies on a number of statistical techniques.

In the simplest case, Impressions needs to create statistically accurate file-system images with default distributions. Hence, a basic functionality required by Impressions is to convert the parameterized distributions into real sample values used to create an instance of a file-system image. Impressions uses random sampling to take a number of independent observations from the respective probability distributions. Wherever applicable, such parameterized distributions provide a highly compact and easy-to-reproduce representation of observed distributions. For cases where standard probability distributions are infeasible, a Monte Carlo method is used.

A user may want to use file system datasets other than the default choice. To enable this, Impressions provides automatic curve-fitting of empirical data.

Impressions also provides the user with the flexibility to specify distributions and constraints on parameter values. One challenge thus is to ensure that multiple constraints specified by the user are resolved consistently. This requires statistical techniques to ensure that the generated file-system images are accurate with respect to both the user-specified constraints and the default distributions.

In addition, the user may want to explore values of file system parameters, not captured in any dataset. For this purpose, Impressions provides support for interpolation and extrapolation of new curves from existing datasets.

Finally, to ensure the accuracy of the generated image, Impressions contains a number of built-in statistical tests, for goodness-of-fit (*e.g.*, Kolmogorov-Smirnov, Chi-Square, and Anderson-Darling), and to estimate er-

ror (*e.g.*, Confidence Intervals, MDCC, and Standard Error). Where applicable, these tests ensure that all curve-fit approximations and internal statistical transformations adhere to the highest degree of statistical rigor desired.

3.3 Creating Valid Metadata

The simplest use of Impressions is to generate file-system images with realistic metadata. This process is performed in two phases: first, the skeletal file-system namespace is created; and second, the namespace is populated with files conforming to a number of file and directory distributions.

3.3.1 Creating File-System Namespace

The first phase in creating a file system is to create the namespace structure or the *directory tree*. We assume that the user specifies the size of the file-system image. The count of files and directories is then selected based on the file system size (if not specified by the user). Depending on the degree of detail desired by the user, each file or directory attribute is selected step by step until all attributes have been assigned values. We now describe this process assuming the highest degree of detail.

To create directory trees, Impressions uses the generative model proposed by Agrawal *et al.* [1] to perform a Monte Carlo simulation. According to this model, new directories are added to a file system one at a time, and the probability of choosing each extant directory as a parent is proportional to $C(d) + 2$, where $C(d)$ is the count of extant subdirectories of directory d . The model explains the creation of the file system namespace, accounting both for the size and count of directories by depth, and the size of parent directories. The input to this model is the total number of directories in the file system. Directory names are generated using a simple iterative counter.

To ensure the accuracy of generated images, we compare the generated distributions (*i.e.*, created using the parameters listed in Table 2), with the desired distributions (*i.e.*, ones obtained from the dataset discussed previously in §3.1). Figure 2 shows in detail the accuracy for each step in the namespace and file creation process. For almost all the graphs, the y-axis represents the percentage of files, directories, or bytes belonging to the categories or bins shown on the x-axis, as the case may be.

Figures 2(a) and 2(b) show the distribution of directories by depth, and directories by subdirectory count, respectively. The y-axis in this case is the percentage of directories at each level of depth in the namespace, shown on the x-axis. The two curves representing the generated and the desired distributions match quite well, indicating good accuracy and reaffirming prior results [1].

3.3.2 Creating Files

The next phase is to populate the directory tree with files. Impressions spends most of the total runtime and effort

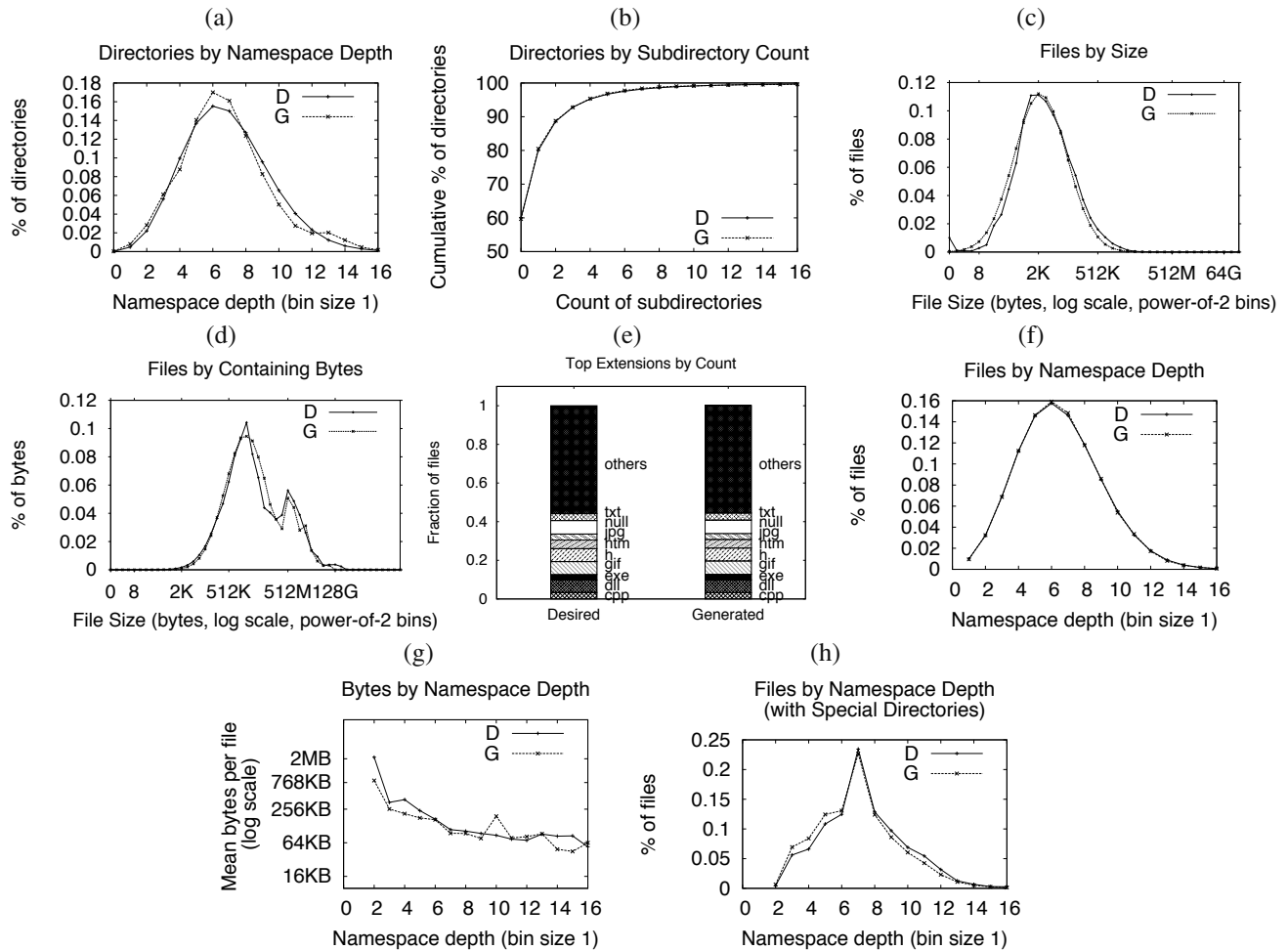


Figure 2: Accuracy of Impressions in recreating file system properties. Shows the accuracy of the entire set of file system distributions modeled by Impressions. D: the desired distribution; G: the generated distribution. Impressions is quite accurate in creating realistic file system state for all parameters of interest shown here. We include a special abscissa for the zero value on graphs having a logarithmic scale.

during this phase, as the bulk of its statistical machinery is exercised in creating files. Each file has a number of attributes such as its size, depth in the directory tree, parent directory, and file extension. Similarly, the choice of the parent directory is governed by directory attributes such as the count of contained subdirectories, the count of contained files, and the depth of the parent directory. Analytical approximations for file system distributions proposed previously [12] guided our own models.

First, for each file, the size of the file is sampled from a hybrid distribution describing file sizes. The body of this hybrid curve is approximated by a lognormal distribution, with a Pareto tail distribution ($k=0.91$, $\chi_m=512\text{MB}$) accounting for the heavy tail of files with size greater than 512 MB. The exact parameter values used for these distributions are listed in Table 2. These parameters were obtained by fitting the respective curves to file sizes obtained from the file-system dataset previously discussed (§3.1). Figure 2(c) shows the accuracy of generating the distribution of files by size. We initially

used a simpler model for file sizes represented solely by a lognormal distribution. While the results were acceptable for files by size (Figure 2(c)), the simpler model failed to account for the distribution of bytes by containing file size; coming up with a model to accurately capture the bimodal distribution of bytes proved harder than we had anticipated. Figure 2(d) shows the accuracy of the hybrid model in Impressions in generating the distribution of bytes. The pronounced double mode observed in the distribution of bytes is a result of the presence of a few large files; an important detail that is otherwise missed if the heavy-tail of file sizes is not accurately accounted for.

Once the file size is selected, we assign the file name and extension. Impressions keeps a list of percentile values for popular file extensions (*i.e.*, top 20 extensions by count, and by bytes). These extensions together account for roughly 50% of files and bytes in a file system ensuring adequate coverage for the important extensions. The remainder of files are given randomly generated three-

Parameter	MDCC
Directory count with depth	0.03
Directory size (subdirectories)	0.004
File size by count	0.04
File size by containing bytes	0.02
Extension popularity	0.03
File count with depth	0.05
Bytes with depth	0.12 MB*
File count w/ depth w/ special dirs	0.06

Table 3: Statistical accuracy of generated images.

Shows average accuracy of generated file-system images in terms of the MDCC (Maximum Displacement of the Cumulative Curves) representing the maximum difference between cumulative curves of generated and desired distributions. Averages are shown for 20 trials. (*) For bytes with depth, MDCC is not an appropriate metric, we instead report the average difference in mean bytes per file (MB). The numbers correspond to the set of graphs shown in Figure 2 and reflect fairly accurate images.

character extensions. Currently filenames are generated by a simple numeric counter incremented on each file creation. Figure 2(e) shows the accuracy of Impressions in creating files with popular extensions by count.

Next, we assign file depth d , which requires satisfying two criteria: the distribution of files with depth, and the distribution of bytes with depth. The former is modeled by a Poisson distribution, and the latter is represented by the mean file sizes at a given depth. Impressions uses a multiplicative model combining the two criteria, to produce appropriate file depths. Figures 2(f) and 2(g) show the accuracy in generating the distribution of files by depth, and the distribution of bytes by depth, respectively.

The final step is to select a parent directory for the file, located at depth $d - 1$, according to the distribution of directories with file count, modeled using an inverse-polynomial of degree 2. As an added feature, Impressions supports the notion of “Special” directories containing a disproportionate number of files or bytes (e.g., “Program Files” folder in the Windows environment). If required, during the selection of the parent directory, a selection bias is given to these special directories. Figure 2(h) shows the accuracy in supporting special directories with an example of a *typical* Windows file system having files in the web cache at depth 7, in Windows and Program Files folders at depth 2, and System files at depth 3.

Table 3 shows the average difference between the generated and desired images from Figure 2 for 20 trials. The difference is measured in terms of the MDCC (Maximum Displacement of the Cumulative Curves). For instance, an MDCC value of 0.03 for directories with depth, implies a *maximum* difference of 3% on an average, between the desired and the generated cumulative distributions. Overall, we find that the models created and used by Impressions for representing various file-

system parameters produce fairly accurate distributions in all the above cases. While we have demonstrated the accuracy of Impressions for the Windows dataset, there is no fundamental restriction limiting it to this dataset. We believe that with little effort, the same level of accuracy can be achieved for any other dataset.

3.4 Resolving Arbitrary Constraints

One of the primary requirements for Impressions is to allow flexibility in specifying file system parameters without compromising accuracy. This means that users are allowed to specify somewhat arbitrary constraints on these parameters, and it is the task of Impressions to resolve them. One example of such a set of constraints would be to specify a large number of files for a small file system, or vice versa, given a file size distribution. Impressions will try to come up with a sample of file sizes that best approximates the desired distribution, while still maintaining the invariants supplied by the user, namely the number of files in the file system and the sum of all file sizes being equal to the file system used space.

Multiple constraints can also be implicit (*i.e.*, arise even in the absence of user-specified distributions). Due to random sampling, different sample sets of the same distribution are not guaranteed to produce exactly the same result, and consequently, the sum of the elements can also differ across samples. Consider the previous example of file sizes again: the sum of all file sizes drawn from a given distribution need not add up to the desired file system size (total used space) each time. More formally, this example is represented by the following set of constraints:

$$\mathcal{N} = \{Constant_1 \vee x : x \in \mathcal{D}_1(x)\}$$

$$\mathcal{S} = \{Constant_2 \vee x : x \in \mathcal{D}_2(x)\}$$

$$\mathcal{F} = \{x : x \in \mathcal{D}_3(x; \mu, \sigma)\}; \left| \sum_{i=0}^{\mathcal{N}} \mathcal{F}_i - \mathcal{S} \right| \leq \beta * \mathcal{S}$$

where \mathcal{N} is the number of files in the file system; \mathcal{S} is the desired file system used space; \mathcal{F} is the set of file sizes; and β is the maximum relative error allowed. The first two constraints specify that \mathcal{N} and \mathcal{S} can be user specified constants or sampled from their corresponding distributions \mathcal{D}_1 and \mathcal{D}_2 . Similarly, \mathcal{F} is sampled from the file size distribution \mathcal{D}_3 . These attributes are further subject to the constraint that the sum of all file sizes differs from the desired file system size by no more than the allowed error tolerance, specified by the user. To solve this problem, we use the following two techniques:

- If the initial sample does not produce a result satisfying all the constraints, we *oversample* additional values of \mathcal{F} from \mathcal{D}_3 , one at a time, until a solution is found, or the oversampling factor α/\mathcal{N} reaches λ (the maximum

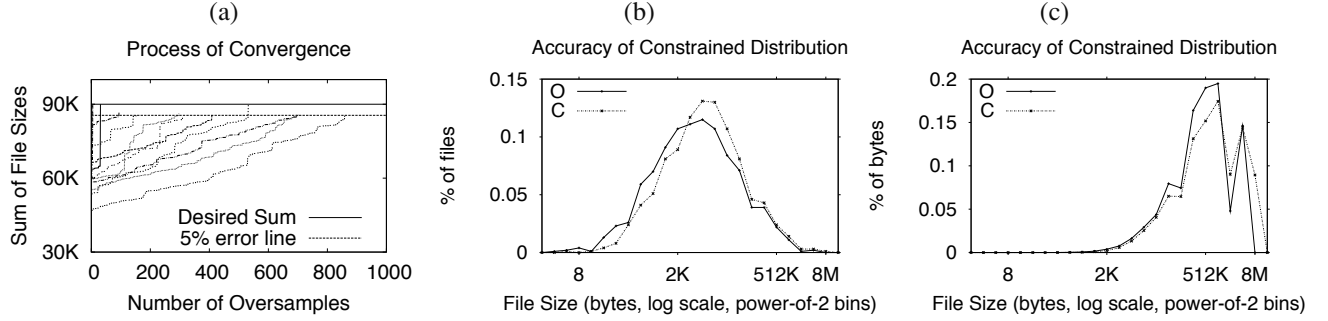


Figure 3: **Resolving Multiple Constraints.** (a) Shows the process of convergence of a set of 1000 file sizes to the desired file system size of 90000 bytes. Each line represents an individual trial. A successful trial is one that converges to the 5% error line in less than 1000 oversamples. (b) Shows the difference between the original distribution of files by size, and the constrained distribution after resolution of multiple constraints in (a). O: Original; C: Constrained. (c) Same as (b), but for distribution of files by bytes instead.

Num. files \mathcal{N}	Sum of file sizes \mathcal{S} (bytes)	File size distribution \mathcal{D}_3	Avg. β Initial	Avg. β Final	Avg. α	Avg. D Count	Avg. D Bytes	Success
1000	30000	($\mu=8.16, \sigma=2.46$)	21.55%	2.04%	5.74%	0.043	0.050	100%
1000	60000	($\mu=8.16, \sigma=2.46$)	20.01%	3.11%	4.89%	0.032	0.033	100%
1000	90000	($\mu=8.16, \sigma=2.46$)	34.35%	4.00%	41.2%	0.067	0.084	90%

Table 4: **Summary of resolving multiple constraints.** Shows average rate and accuracy of convergence after resolving multiple constraints for different values of desired file system size. β : % error between the desired and generated sum, α : % of oversamples required, D is the test statistic for the K-S test representing the maximum difference between generated and desired empirical cumulative distributions. Averages are for 20 trials. Success is the number of trials having final $\beta \leq 5\%$, and D passing the K-S test.

oversampling factor). α is the count of extra samples drawn from \mathcal{D}_3 . Upon reaching λ without finding a solution, we discard the current sample set and start over.

- The number of elements in \mathcal{F} during the oversampling stage is $\mathcal{N} + \alpha$. For every oversampling, we need to find if there exists \mathcal{F}_{Sub} , a subset of \mathcal{F} with \mathcal{N} elements, such that the sum of all elements of \mathcal{F}_{Sub} (file sizes) differs from the desired file system size by no more than the allowed error. More formally stated, we find if:

$$\exists \mathcal{F}_{Sub} = \{\mathcal{X} : \mathcal{X} \subseteq \mathbb{P}(F), |\mathcal{X}| = \mathcal{N}, |\mathcal{F}| = \mathcal{N} + \alpha,$$

$$| \sum_{i=0}^{\mathcal{N}} \mathcal{X}_i - \mathcal{S} | \leq \beta * \mathcal{S}, \alpha \in \mathbb{N} \wedge \frac{\alpha}{\mathcal{N}} \leq \lambda \}$$

The problem of resolving multiple constraints as formulated above, is a variant of the more general “Subset Sum Problem” which is NP-complete [8]. Our solution is thus an approximation algorithm based on an existing $O(n \log n)$ solution [37] for the Subset Sum Problem.

The existing algorithm has two phases. The first phase randomly chooses a solution vector which is valid (the sum of elements is less than the desired sum), and maximal (adding any element not already in the solution vector will cause the sum to exceed the desired sum). The second phase performs *local improvement*: for each element in the solution, it searches for the largest element not in the current solution which, if replaced with the current element, would reduce the difference between the desired and current sums. The solution vector is updated if such an element is found, and the algorithm proceeds

with the next element, until all elements are compared.

Our problem definition and the modified algorithm differ from the original in the following ways:

- First, in the original problem, there is no restriction on the number of elements in the solution subset \mathcal{F}_{Sub} . In our case, \mathcal{F}_{Sub} can have exactly \mathcal{N} elements. We modify the first phase of the algorithm to set the initial \mathcal{F}_{Sub} as the first random permutation of \mathcal{N} elements selected from \mathcal{F} such that their sum is less than \mathcal{S} .
- Second, the original algorithm either finds a solution or terminates without success. We use an increasing sample size after each oversampling to reduce the error, and allow the solution to converge.
- Third, it is not sufficient for the elements in \mathcal{F}_{Sub} to have a numerical sum close to the desired sum \mathcal{S} , but the distribution of the elements must also be close to the original distribution in \mathcal{F} . A goodness-of-fit test at the end of each oversampling step enforces this requirement. For our example, this ensures that the set of file sizes generated after resolving multiple constraints still follow the original distribution of file sizes.

The algorithm terminates successfully when the difference between the sums, and between the distributions, falls below the desired error levels. The success of the algorithm depends on the choice of the desired sum, and the *expected* sum (the sum due to the choice of parameters, e.g., μ and σ); the farther the desired sum is from the expected sum, the lesser are the chances of success.

Consider an example where a user has specified a desired file system size of 90000 bytes, a lognormal file

size distribution ($\mu=8.16$, $\sigma=2.46$), and 1000 files. Figure 3(a) shows the convergence of the sum of file sizes in a sample set obtained with this distribution. Each line in the graph represents an independent trial, starting at a y-axis value equal to the sum of its initially sampled file sizes. Note that in this example, the initial sum differs from the desired sum by more than a 100% in several cases. The x-axis represents the number of extra iterations (*oversamples*) performed by the algorithm. For a trial to succeed, the sum of file sizes in the sample must converge to within 5% of the desired file system size. We find that in most cases λ ranges between 0 and 0.1 (i.e., less than 10% oversampling); and in almost all cases, $\lambda \leq 1$.

The distribution of file sizes in \mathcal{F}_{Sub} must be close to the original distribution in \mathcal{F} . Figure 3(b) and 3(c) show the difference between the original and constrained distributions for file sizes (for files by size, and files by bytes), for one successful trial from Figure 3(a). We choose these particular distributions as examples throughout this paper for two reasons. First, file size is an important parameter, so we want to be particularly thorough in its accuracy. Second, getting an accurate shape for the bimodal curve of files by bytes presents a challenge for Impressions; once we get our techniques to work for this curve, we are fairly confident of its accuracy on simpler distributions.

We find that Impressions resolves multiple constraints to satisfy the requirement on the sum, while respecting the original distributions. Table 4 gives the summary for the above example of file sizes for different values of the desired file system size. The expected sum of 1000 file sizes, sampled as specified in the table, is close to 60000. Impressions successfully converges the initial sample set to the desired sum with an average oversampling rate α less than 5%. The average difference between the desired and achieved sum β is close to 3%. The constrained distribution passes the two-sample K-S test at the 0.05 significance level, with the difference between the two distributions being fairly small (the D statistic of the K-S test is around 0.03, which represents the maximum difference between two empirical cumulative distributions).

We repeat the above experiment for two more choices of file system sizes, one lower than the expected mean (30K), and one higher (90K); we find that even when the desired sum is quite different from the expected sum, our algorithm performs well. Only for 2 of the 20 trials in the 90K case, did the algorithm fail to converge. For these extreme cases, we drop the initial sample and start over.

3.5 Interpolation and Extrapolation

Impressions requires knowledge of the distribution of file system parameters necessary to create a valid image. While it is tempting to imagine that Impressions has

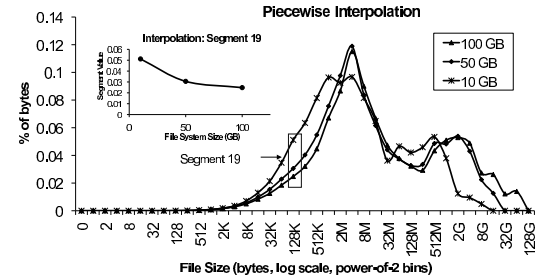


Figure 4: **Piecewise Interpolation of File Sizes.** Piecewise interpolation for the distribution of files with bytes, using file systems of 10 GB, 50 GB and 100 GB. Each power-of-two bin on the x-axis is treated as an individual segment for interpolation (inset). Final curve is the composite of all individual interpolated segments.

Distribution	FS Region (I/E)	D Statistic	K-S Test (0.05)
File sizes by count	75GB (I)	0.054	passed
File sizes by count	125GB (E)	0.081	passed
File sizes by bytes	75GB (I)	0.105	passed
File sizes by bytes	125GB (E)	0.105	passed

Table 5: **Accuracy of interpolation and extrapolation.** Impressions produces accurate curves for file systems of size 75 GB and 125 GB, using interpolation (I) and extrapolation (E), respectively.

perfect knowledge about the nature of these distributions for all possible values and combinations of individual parameters, it is often impossible.

First, the empirical data is limited to what is observed in any given dataset and may not cover the entire range of possible values for all parameters. Second, even with an exhaustive dataset, the user may want to explore regions of parameter values for which no data point exists, especially for “what if” style of analysis. Third, from an implementation perspective, it is more efficient to maintain compact representations of distributions for a few sample points, instead of large sets of data. Finally, if the empirical data is statistically insignificant, especially for outlying regions, it may not serve as an accurate representation. Impressions thus provides the capability for interpolation and extrapolation from available data and distributions.

Impressions needs to generate complete new curves from existing ones. To illustrate our procedure, we describe an example of creating an interpolated curve; extensions to extrapolation are straightforward. Figure 4 shows how Impressions uses *piece-wise interpolation* for the distribution of files with containing bytes. In this example, we start with the distribution of file sizes for file systems of size 10 GB, 50 GB and 100 GB, shown in the figure. Each power-of-two bin on the x-axis is treated as an individual *segment*, and the available data points within each segment are used as input for piece-wise interpolation; the process is repeated for all segments of the curve. Impressions combines the individual interpolated segments to obtain the complete interpolated curve.

To demonstrate the accuracy of our approach, we in-

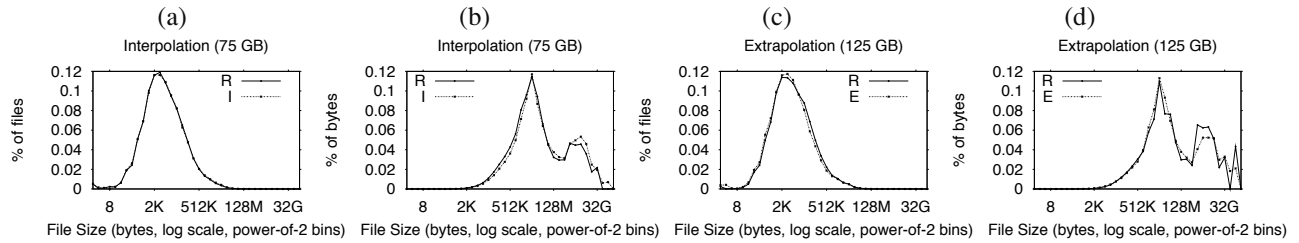


Figure 5: Accuracy of Interpolation and Extrapolation. Shows results of applying piece-wise interpolation to generate file size distributions (by count and by bytes), for file systems of size 75 GB (a and b, respectively), and 125 GB (c and d, respectively).

terpolate and extrapolate file size distributions for file systems of sizes 75 GB and 125 GB, respectively. Figure 5 shows the results of applying our technique, comparing the generated distributions with actual distributions for the file system sizes (we removed this data from the dataset used for interpolation). We find that the simpler curves such as Figure 5(a) and (c) are interpolated and extrapolated with good accuracy. Even for more challenging curves such as Figure 5(b) and (d), the results are accurate enough to be useful. Table 5 contains the results of conducting K-S tests to measure the goodness-of-fit of the generated curves. All the generated distributions passed the K-S test at the 0.05 significance level.

3.6 File Content

Actual file content can have substantial impact on the performance of an application. For example, Postmark [24], one of the most popular file system benchmarks, tries to simulate an email workload, yet it pays scant attention to the organization of the file system, and is completely oblivious of the file data. Postmark fills all the “email” files with the same data, generated using the same random seed. The evaluation results can range from misleading to completely inaccurate, for instance in the case of content-addressable storage (CAS). When evaluating a CAS-based system, the disk-block traffic and the corresponding performance will depend only on the unique content – in this case belonging to the largest file in the file system. Similarly, performance of Desktop Search and Word Processing applications is sensitive to file content.

In order to generate representative file content, Impressions supports a number of options. For human-readable files such as `.txt`, `.html` files, it can populate file content with random permutations of symbols and words, or with more sophisticated word-popularity models. Impressions maintains a list of the relative popularity of the most popular words in the English language, and a Monte Carlo simulation generates words for file content according to this model. However, the distribution of word popularity is heavy-tailed; hence, maintaining an exhaustive list of words slows down content generation. To improve performance, we use a word-length fre-

quency model [43] to generate the long tail of words, and use the word-popularity model for the body alone. The user has the flexibility to select either one of the models in entirety, or a specific combination of the two. It is also relatively straightforward to add extensions in the future to generate more nuanced file content. An example of such an extension is one that carefully controls the degree of content similarity across files.

In order to generate content for typed files, Impressions either contains enough information to generate valid file headers and footers itself, or calls into a third-party library or software such as `Id3v2` [31] for `mp3`; `GraphApp` [18] for `gif`, `jpeg` and other image files; `Mplayer` [28] for `mpeg` and other video files; `asciidoc` for `html`; and `ascii2pdf` for `PDF` files.

3.7 Disk Layout and Fragmentation

To isolate the effects of file system content, Impressions can measure the degree of on-disk fragmentation, and create file systems with user-defined degree of fragmentation. The extent of fragmentation is measured in terms of *layout score* [44]. A layout score of 1 means all files in the file system are laid out optimally on disk (*i.e.*, all blocks of any given file are laid out consecutively one after the other), while a layout score of 0 means that no two blocks of any file are adjacent to each other on disk.

Impressions achieves the desired degree of fragmentation by issuing pairs of temporary file create and delete operations, during creation of regular files. When experimenting with a file-system image, Impressions gives the user complete control to specify the overall layout score. In order to determine the on-disk layout of files, we rely on the information provided by debugfs. Thus currently we support layout measurement only for `Ext2` and `Ext3`. In future work, we will consider several alternatives for retrieving file layout information across a wider range of file systems. On Linux, the `FIBMAP` and `FIEMAP ioctl()`s are available to map a logical block to a physical block [23]. Other file system-specific methods exist, such as the `XFS_IOC_GETBMAP ioctl` for `XFS`.

The previous approach however does not account for differences in fragmentation strategies across file systems. Impressions supports an alternate specification

FS distribution (Default)	Time taken (seconds)	
	<i>Image₁</i>	<i>Image₂</i>
Directory structure	1.18	1.26
File sizes distribution	0.10	0.28
Popular extensions	0.05	0.13
File with depth	0.064	0.29
File and bytes with depth	0.25	0.70
File content (Single-word)	0.53	1.44
On-disk file/dir creation	437.80	1394.84
Total time	473.20 (8 mins)	1826.12 (30 mins)
File content (Hybrid model)	791.20	—
Layout score (0.98)	133.96	—

Table 6: Performance of Impressions. *Shows time taken to create file-system images with break down for individual features. Image₁: 4.55 GB, 20000 files, 4000 dirs. Image₂: 12.0 GB, 52000 files, 4000 dirs. Other parameters are default. The two entries for additional parameters are shown only for Image₁ and represent times in addition to default times.*

for the degree of fragmentation wherein it runs a pre-specified workload and reports the resulting layout score. Thus if a file system employs better strategies to avoid fragmentation, it is reflected in the final layout score after running the fragmentation workload.

There are several alternate techniques for inducing more realistic fragmentation in file systems. Factors such as burstiness of I/O traffic, out-of-order writes and inter-file layout are currently not accounted for; a companion tool to Impressions for carefully creating fragmented file systems will thus be a good candidate for future research.

3.8 Performance

In building Impressions, our primary objective was to generate realistic file-system images, giving top priority to accuracy, instead of performance. Nonetheless, Impressions does perform reasonably well. Table 6 shows the breakdown of time taken to create a default file-system image of 4.55 GB. We also show time taken for some additional features such as using better file content, and creating a fragmented file system. Overall, we find that Impressions creates highly accurate file-system images in a reasonable amount of time and thus is useful in practice.

4 Case Study: Desktop Search

In this section, we use Impressions to evaluate desktop searching applications. Our goals for this case study are two-fold. First, we show how simple it is to use Impressions to create either representative images or images across which a single parameter is varied. Second, we show how future evaluations should report the settings of Impressions so that results can be easily reproduced.

We choose desktop search for our case study because its performance and storage requirements depend not only on the file system size and structure, but also on the

type of files and the actual content within the files. We evaluate two desktop search applications: open-source Beagle [5] and Google’s Desktop for Linux (GDL) [16]. Beagle supports a large number of file types using 52 search-filters; it provides several indexing options, trading performance and index size with the quality and feature-richness of the index. Google Desktop does not provide as many options: a web interface allows users to select or exclude types of files and folder locations for searching, but does not provide any control over the type and quality of indexing.

4.1 Representative Images

Developers of data-intensive applications frequently need to make assumptions about the properties of file-system images. For example, file systems and applications can often be optimized if they know properties such as the relative proportion of meta-data to data in representative file systems. Previously, developers could infer these numbers from published papers [1, 12, 41, 42], but only with considerable effort. With Impressions, developers can simply create a sample of representative images and directly measure the properties of interest.

Table 6 lists assumptions we found in GDL and Beagle limiting the search indexing to partial regions of the file system. However, for the representative file systems in our data set, these assumptions omit large portions of the file system. For example, GDL limits its index to only those files less than ten directories deep; our analysis of typical file systems indicates that this restriction causes 10% of all files to be missed. We believe that instead of arbitrarily specifying hard values, application designers should experiment with Impressions to find acceptable choices.

We note that Impressions is useful for discovering these application assumptions and for isolating performance anomalies that depend on the file-system image. Isolating the impact of different file systems feature is easy using Impressions: evaluators can use Impressions to create file-system images in which only a single parameter is varied, while all other characteristics are carefully controlled.

This type of discovery is clearly useful when one is using closed-source code, such as GDL. For example, we discovered the GDL limitations by constructing file-system images across which a single parameter is varied (*e.g.*, file depth and file size), measuring the percentage of indexed files, and noticing precipitous drops in this percentage. This type of controlled experimentation is also useful for finding non-obvious performance interactions in open-source code. For instance, Beagle uses the *inotify* mechanism [22] to track each directory for change; since the default Linux kernel provides 8192 watches, Beagle resorts to manually crawling the directo-

App	Parameter & Value	Comment on Validity
GDL	File content < 10 deep	10% of files and 5% of bytes > 10 deep (content in deeper namespace is growing)
GDL	Text file sizes < 200 KB	13% of files and 90% of bytes > 200 KB
Beagle	Text file cutoff < 5 MB	0.13% of files and 71% of bytes > 5 MB
Beagle	Archive files < 10 MB	4% of files and 84% of bytes > 10 MB
Beagle	Shell scripts < 20 KB	20% of files and 89% of bytes > 20 KB

Figure 6: **Debunking Application Assumptions.** Examples of assumptions made by Beagle and GDL, along with details of the amount of file-system content that is not indexed as a consequence.

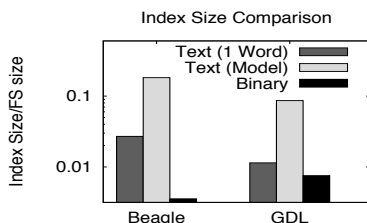


Figure 7: **Impact of file content.** Compares Beagle and GDL index time and space for wordmodels and binary files. Google has a smaller index for wordmodels, but larger for binary. Uses Impressions default settings, with FS size 4.55 GB, 20000 files, 4000 dirs.

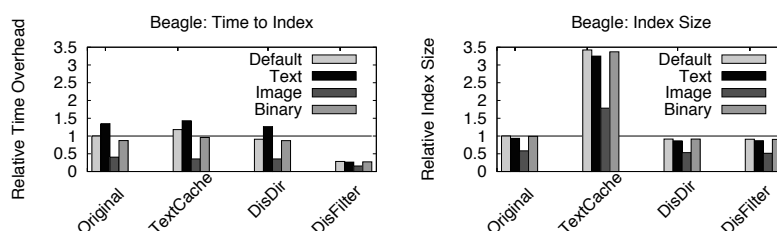


Figure 8: **Reproducible images: impact of content.** Using Impressions to make results reproducible for benchmarking search. Vertical bars represent file systems created with file content as labeled. The Default file system is created using Impressions default settings, and file system size 4.55 GB, 20000 files, 4000 dirs. Index options: Original – default Beagle index. TextCache – build text-cache of documents used for snippets. DisDir – don't add directories to the index. DisFilter – disable all filtering of files, only index attributes.

ries once their count exceeds 8192. This deterioration in performance can be easily found by creating file-system images with varying numbers of directories.

4.2 Reproducible Images

The time spent by desktop search applications to crawl a file-system image is significant (*i.e.*, hours to days); therefore, it is likely that different developers will innovate in this area. In order for developers to be able to compare their results, they must be able to ensure they are using the same file-system images. Impressions allows one to precisely control the image and report the parameters so that the exact same image can be reproduced.

For desktop search, the type of files (*i.e.*, their extensions) and the content of files has a significant impact on the time to build the index and its size. We imagine a scenario in which the Beagle and GDL developers wish to compare index sizes. To make a meaningful comparison, the developers must clearly specify the file-system image used; this can be done easily with Impressions by reporting the size of the image, the distributions listed in Table 2, the word model, disk layout, and the random seed. We anticipate that most benchmarking will be done using mostly default values, reducing the number of Impressions parameters that must be specified.

An example of the reporting needed for reproducible results is shown in Figure 7. In these experiments, all distributions of the file system are kept constant, but only either text files (containing either a single word or with the default word model) or binary files are created. These experiments illustrate the point that file content signif-

icantly affects the index size; if two systems are compared using different file content, obviously the results are meaningless. Specifically, different file types change even the relative ordering of index size between Beagle and GDL: given text files, Beagle creates a larger index; given binary files, GDL creates a larger index.

Figures 8 gives an additional example of reporting Impressions parameters to make results reproducible. In these experiments, we discuss a scenario in which different developers have optimized Beagle and wish to meaningfully compare their results. In this scenario, the original Beagle developers reported results for four different images: the default, one with only text files, one with only image files, and one with only binary files. Other developers later create variants of Beagle: *TextCache* to display a small portion of every file alongside a search hit, *DisDir* to disable directory indexing, and *DisFilter* to index only attributes. Given the reported Impressions parameters, the variants of Beagle can be meaningfully compared to one another.

In summary, Impressions makes it extremely easy to create both controlled and representative file-system images. Through this brief case study evaluating desktop search applications, we have shown some of the advantages of using Impressions. First, Impressions enables developers to tune their systems to the file system characteristics likely to be found in their target user populations. Second, it enables developers to easily create images where one parameter is varied and all others are carefully controlled; this allows one to assess the impact of a single parameter. Finally, Impressions enables different developers to ensure they are all comparing the

same image; by reporting Impressions parameters, one can ensure that benchmarking results are reproducible.

5 Related Work

We discuss previous research in four areas related to file system benchmarking and usage of file system metadata.

First, Impressions enables file system measurement studies to be put into practice. Besides the metadata studies on Windows workstations [1, 12], previous work in non-Windows environment includes Satyanarayanan's study of a Digital PDP-10 [41], Irlam's and Mullender's studies of Unix systems [21, 29], and the study of HP-UX systems at Hewlett-Packard [42]. These studies provide valuable data for designers of file systems and related software, and can be easily incorporated in Impressions.

Second, several models have been proposed to explain observed file-system phenomena. Mitzenmacher proposed a generative model, called the Recursive Forest File model [27] to explain the behavior of file size distributions. The model accounts for the hybrid distribution of file sizes with a lognormal body and Pareto tail. Downey's Multiplicative File Size model [13] is based on the assumption that new files are created by using older files as templates e.g., by copying, editing or filtering an old file. The size of the new file in this model is given by the size of the old file multiplied by an independent factor. These models provide an intuitive understanding of the underlying phenomena, and are also easier for computer simulation. In future, Impressions can be enhanced by incorporating more such models.

Third, a number of tools and techniques have been proposed to improve the state of the art of benchmarking. Chen and Patterson proposed a "self-scaling" benchmark that scales with the I/O system being evaluated, to stress the system in meaningful ways [6]. TBBT is a NFS trace replay tool that derives the file-system image underlying a trace [50]. It extracts the file system hierarchy from a given trace in depth-first order and uses that during initialization for a subsequent trace replay. While this ensures a consistent file-system image for replay, it does not solve the more general problem of creating accurately controlled images for all types of file system benchmarking. The Auto-Pilot tool [48] provides an infrastructure for running tests and analysis tools to automate the benchmarking process.

Finally, workload is an important piece of the benchmarking puzzle. The SynRGen file reference generator by Ebling and Satyanarayan [14] generates synthetic equivalents for real file system users. The *volumes* or images in their work make use of simplistic assumptions about the file system distributions as their focus is on user access patterns. Roselli *et al.* collected dynamic file system usage patterns in UNIX and Windows NT environ-

ments and studied file system access behavior [39]. Recent work on file system workloads includes a study of network file system usage at NetApp [25].

6 Conclusion

File system benchmarking is in a state of disarray. One key aspect of this problem is generating realistic file-system state, with due emphasis given to file-system metadata and file content. To address this problem, we develop Impressions, a statistical framework to generate realistic and configurable file-system images. Impressions provides the user flexibility in selecting a comprehensive set of file system parameters, while seamlessly ensuring accuracy of the underlying images, serving as a useful platform for benchmarking.

In our experience, we find Impressions easy to use and well suited for a number of tasks. It enables application developers to tune their systems to the file system characteristics likely found in their target users. Impressions also makes it feasible to compare performance of systems by standardizing and reporting all used parameters, a requirement necessary for benchmarking. We believe Impressions will prove to be a valuable tool for system developers and users alike; we intend to release it for public use in the near future. Please check <http://www.cs.wisc.edu/adsl/Software/Impressions/> to obtain a copy.

7 Acknowledgments

We are grateful to Bill Bolosky for providing us with a copy of the five-year metadata dataset from Microsoft. Lakshmi Bairavasundaram provided many useful discussions and gave valuable comments on earlier drafts of this paper. Finally, we would like to thank Valerie Aurora Henson (our shepherd) and the anonymous reviewers for their excellent feedback and comments.

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, as well as by generous donations from Network Appliance and Sun Microsystems. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

- [1] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A Five-Year Study of File-System Metadata. In *FAST '07*, San Jose, CA, February 2007.
- [2] D. Anderson and J. Chase. Fstress: A flexible network file service benchmark. In *TR, Duke University*, May 2002.

- [3] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan. But-tress: A toolkit for flexible and high fidelity I/O benchmarking. In *FAST '04*, San Francisco, CA, April 2004.
- [4] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *SOSP '91*, pages 198–212, Pacific Grove, CA, October 1991.
- [5] Beagle Project. Beagle Desktop Search. <http://www.beagle-project.org/>.
- [6] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation—Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *SIGMETRICS '93*, pages 1–12, Santa Clara, CA, May 1993.
- [7] J. Cipar, M. D. Corner, and E. D. Berger. Tfs: a transparent file system for contributory storage. In *FAST '07*, pages 28–28, Berkeley, CA, USA, 2007. USENIX Association.
- [8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition, 2001. 35.5: The subset-sum problem.
- [9] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. *SIGOPS Oper. Syst. Rev.*, 36, 2002.
- [10] L. P. Cox and B. D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 120–132, New York, NY, USA, 2003. ACM.
- [11] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *OSDI '94*, Monterey, CA, November 1994.
- [12] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the 1999 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 59–70, Atlanta, GA, May 1999.
- [13] A. B. Downey. The structural cause of file size distributions. In *Ninth MASCOTS '01*, Los Alamitos, CA, USA, 2001.
- [14] M. R. Ebling and M. Satyanarayanan. Synrgen: an extensible file reference generator. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, New York, NY, 1994.
- [15] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. *ACM Trans. Comput. Syst.*, 20(1):1–24, 2002.
- [16] Google Corp. Google Desktop for Linux. <http://desktop.google.com/linux/index.html>.
- [17] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In *OSDI '99: Third symposium on Operating Systems Design and Implementation*, 1999.
- [18] GraphApp. GraphApp Toolkit. <http://enchantia.com/software/graphapp/>.
- [19] S. D. Gribble, G. S. Manku, D. S. Roselli, E. A. Brewer, T. J. Gibson, and E. L. Miller. Self-similarity in file systems. In *Proceedings of the 1998 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 141–150, Madison, WI, June 1998.
- [20] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. Physical File System Backup. In *OSDI '99*, New Orleans, LA, February 1999.
- [21] G. Irlam. Unix file size survey – 1993. Available at <http://www.base.com/gordon/ufs93.html>.
- [22] John McCutchan and Robert Love. inotify for linux. <http://www.linuxjournal.com/article/8478>.
- [23] Jonathan Corbet. LWN Article: SEEK_HOLE or FIEMAP? <http://lwn.net/Articles/260795/>.
- [24] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.
- [25] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2008.
- [26] M. P. Mesnier, M. Wachs, R. R. Sambasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. O'Hallaron. trace: parallel trace replay with approximate causal events. In *FAST '07*, San Jose, CA, February 2007.
- [27] M. Mitzenmacher. Dynamic models for file sizes and double pareto distributions. In *Internet Mathematics*, 2002.
- [28] Mplayer. The MPlayer movie player. <http://www.mplayerhq.hu/>.
- [29] S. J. Mullender and A. S. Tanenbaum. Immediate files. *Software—Practice and Experience*, 14(4):365–368, April 1984.
- [30] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, pages 174–187, Banff, Canada, October 2001.
- [31] Myers Carpenter. Id3v2: A command line editor for id3v2 tags. <http://id3v2.sourceforge.net/>.
- [32] NIST. Text retrieval conference (trec) datasets. <http://trec.nist.gov/data>, 2007.
- [33] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *SOSP '85*, pages 15–24, Orcas Island, WA, December 1985.
- [34] Y. Padoleau and O. Ridoux. A logic file system. In *USENIX Annual Technical Conference*, San Antonio, TX, June 2003.
- [35] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88*, pages 109–116, Chicago, IL, June 1988.
- [36] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*, pages 206–220, Brighton, UK, October 2005.
- [37] B. Przydatek. A Fast Approximation Algorithm for the Subset-sum Problem. *International Transactions in Operational Research*, 9(4):437–459, 2002.
- [38] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *FAST '02*, pages 14–29, Monterey, CA, January 2002.
- [39] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *USENIX '00*, pages 41–54, San Diego, CA, June 2000.
- [40] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *SOSP '01*, Banff, Canada, October 2001.
- [41] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*, pages 96–108, Pacific Grove, CA, December 1981.
- [42] T. F. Sienknecht, R. J. Friedrich, J. J. Martinka, and P. M. Friedenbach. The implications of distributed data in a commercial environment on the design of hierarchical storage management. *Performance Evaluation*, 20(1–3):3–25, May 1994.
- [43] B. Sigurd, M. Eeg-Olofsson, and J. van de Weijer. Word length, sentence length and frequency – Zipf revisited. *Studia Linguistica*, 58(1):37–52, 2004.
- [44] K. Smith and M. I. Seltzer. File System Aging. In *Proceedings of the 1997 Sigmetrics Conference*, Seattle, WA, June 1997.
- [45] SNIA. Storage network industry association: Iotta repository. <http://iota.snia.org>, 2007.
- [46] S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, W. Ziskind, and A. Krishnamurthy. Segank: A Distributed Mobile Storage System. In *FAST '04*, pages 239–252, San Francisco, CA, April 2004.
- [47] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–16, Berkeley, CA, USA, 2008. USENIX Association.
- [48] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A platform for system software benchmarking. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, Anaheim, CA, April 2005.
- [49] Z. Zhang and K. Ghose. yfs: A journaling file system design for handling large data sets with reduced seeking. In *FAST '03*, pages 59–72, Berkeley, CA, USA, 2003. USENIX Association.
- [50] N. Zhu, J. Chen, and T.-C. Chiueh. Tbbt: scalable and accurate trace replay for file server evaluation. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 24–24, Berkeley, CA, USA, 2005. USENIX Association.

Capture, conversion, and analysis of an intense NFS workload

Eric Anderson, HP Labs <eric.anderson4@hp.com>

Abstract

We describe methods to capture, convert, store and analyze NFS workloads that are 20-100× more intense, in terms of operations/day, than any previously published. We describe three techniques that improve capture performance by up to 10× over previous techniques. For conversion, we use a general-purpose format that is both highly space efficient and provides efficient access to the trace data. For analysis, we describe a number of techniques adopted from the database community and some new techniques that facilitate analysis of very large traces. We also describe a number of guidelines for trace collection that should prove useful to future practitioners. Finally, we analyze a commercial feature animation (movie) rendering workload using these techniques and discuss the characteristics of the workload. Our implementation of these techniques is available as open source and the exact anonymized datasets we analyze are available for free download.

1 Introduction

Storage tracing and analysis have a long history. Some of the earliest filesystem traces were captured in 1985 [26], and there has been intermittent tracing effort since then, summarized by Leung [20]. Storage traces are analyzed to find properties that future systems should support or exploit, and as input to simulators and replay tools to explore system performance with real workloads.

One of the problems with trace analysis is that old traces inherently have to be scaled up to be used for evaluating newer storage systems because the underlying performance of the newer systems has increased. Therefore the community benefits from regularly capturing new traces from multiple sources, and, if possible, traces that put a heavy load on the storage system, reducing the need to scale the workload.

Most traces, since they are captured by academics, are captured in academic settings. This means that the workloads captured are somewhat comparable, but it also means that commercial workloads are under-represented. Microsoft is working to correct this by capturing commer-

cial enterprise traces from their internal servers [23]. Our work focuses on commercial NFS [25, 6, 28] workloads, in particular from a feature animation (movie) company, whose name remains blinded as part of the agreement to publish the traces. The most recent publically available NFS traces that we are aware of were collected in 2003 by Ellard [13]. Our 2003 and 2007 traces [4] provide recent NFS traces for use by the community.

One difference between our traces and other ones is the data rates that we measured. Our 2003 client traces saw about 750 million operations per day. In comparison, the 2003 Ellard traces saw a peak of about 125 million NFS operations per day, and the 2007 Leung traces [20] saw a peak of 19 million CIFS operations/day. Our 2007 traces saw about 2.4 billion operations/day. This difference required us to develop and adopt new techniques to capture, convert, and analyze the traces.

Since our traces were captured in such a different environment than prior traces, we limit our comparisons to their workloads, and we do not attempt to make any claims about trends. We believe that unless we, as a community, collect traces from hundreds of different sites, we will not have sufficient data to make claims stronger than “this workload is different from other ones in these ways.” In fact, we make limited comparison of the trends between our 2003 and 2007 traces for similar reasons. The underlying workload changed as the rendering techniques improved to generate higher quality output, the operating system generating the requests changed, the NFS protocol version changed, and the configuration of the clients changed because of standard technology trends.

The process of understanding a workload involves four main steps, as shown in Figure 1. Our tools for these steps are shown in italics for each step, as well as some traditional tools. The first step is capturing the workload, usually as some type of trace. The second step is conversion, usually from some raw format into a format designed for analysis. The third step is analysis to reduce the huge amount of converted data to something manageable. Alternately, this step is a simulation or replay to explore some new system architecture. Finally the fourth step is to generate graphs or textual reports from the output of the analysis or simulation.

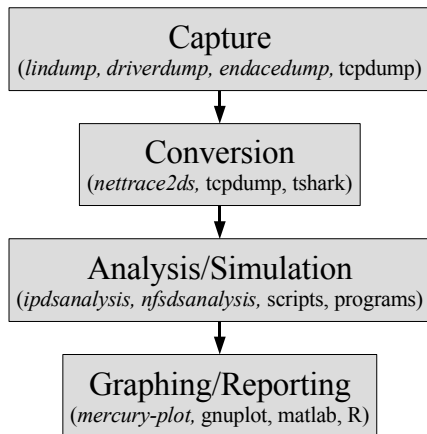


Figure 1: Overall process; our tools are shown in italics, traditional tools after them.

Our work has five main contributions:

1. The development of techniques for lossless raw packet capture up to 5Gb/s, and with recent hardware improvements, likely to 10Gb/s. These techniques are applicable to anyone wanting to capture a network storage service such as NFS, CIFS, or iSCSI.
2. A series of guidelines for the conversion and storage of the traces. Many of these guidelines are things that we wish we had known when we were converting our traces. We used DataSeries [2] to store the traces, but our guidelines are general.
3. Improved techniques for analyzing very large traces that allow us to look at the burstiness in workloads, and an examination of how the long averaging intervals in prior analysis can obscure workload properties.
4. The analysis of an intense NFS workload demonstrating that our techniques are successful.
5. The agreement with the animation company to allow the roughly 100 billion operation anonymized traces to be published, along with the complete set of tools to perform all the analysis presented in this paper and to generate the graphs. Other researchers can build on our tools for further analysis, and use the traces in simulation studies.

We examine related work in Section 2. We describe our capture techniques in Section 3, followed by the conversion in Section 4. We describe our adopted and new analysis techniques in Section 5 and use them to analyze the workload in Section 6. Finally we conclude in Section 7.

2 Related work

The two closest pieces of related work are Ellard’s NFS study [12, 13], and Leung’s 2007 CIFS study [20]. These papers also summarize the earlier decade of filesystem tracing, so we refer interested readers to those papers. Ellard et al. captured NFS traces from a number of Digital UNIX and NetApp servers on the Harvard campus, analyzed the traces and presented new results looking at the sequentiality of the workload, and comparing his results to earlier traces. Ellard made his tools available, so we initially considered building on top of them, but quickly discovered that our workload was so much more intense that his tools would be insufficient, and so ended up building our own. We later translated those tools and traces into DataSeries, and found our version was about $100\times$ faster on a four core machine and used $25\times$ less CPU time for analysis. Our 2003 traces were about $25\times$ more intense than Ellard’s 2001 traces, and about $6\times$ more intense than Ellard’s 2003 traces.

Leung et al. traced a pair of NetApp servers on their campus. Since the clients were entirely running the Windows operating system, his traces were of CIFS data, and so he used the Wireshark tools [31] to convert the traces. Leung’s traces were of comparable intensity to Ellard’s traces, and they noted that they had some small packet drops during high load as they just used tcpdump for capture. Leung identified and extensively analyzed complicated sequentiality patterns. Our 2007 traces were about $95\times$ more intense than Leung’s traces, as they saw a peak of 19.1 million operations/day and we saw an average of about 1.8 billion. This comparison is slightly misleading as NFS tends to have more operations than CIFS because NFS is a stateless protocol.

Tcpdump [30] is the tool that almost all researchers describe using to capture packet traces. We tried using tcpdump, but experienced massive packet loss using it in 2003, and so developed new techniques. For compatibility, we used the pcap file format, originally developed for tcpdump, for our raw captured data. When we captured our second set of traces in 2007, we needed to capture at even higher rates, and we used a specialized capture card. We wrote new capture software using techniques we had developed in 2003 to allow us to capture above 5Gb/s.

Tcpdump also includes limited support for conversion of NFS packets. Wireshark [31] provides a graphical interface to packet analysis, and the tshark variant provides conversion to text. We were not aware of Wireshark at the time of our first capture, and we simply adjusted our earlier tools when we did our 2007 tracing. We may consider using the Wireshark converter in the future, provided we can make it run much faster. Running tshark on a small 2

million packet capture took about 45 seconds whereas our converter ran in about 5 seconds. Given conversion takes 2-3 days for a 5 day trace, we can not afford conversion to slow down by a factor of $9\times$.

Some of the analysis techniques we use are derived from the database community, namely the work on cubes [16] and approximate quantiles [22]. We considered using a standard SQL database for our storage and analysis, but abandoned that quickly because a database that can hold 100 billion rows is very expensive. We do use SQL databases for analysis and graphing once we have reduced the data size down to a few million rows using our tools.

3 Raw packet capture

The first stage in analyzing an NFS workload is capturing the data. There are three places that the workload could be captured: the client, the server, or the network. Capturing the workload on the clients is very parallel, but is difficult to configure and can interfere with the real workload. Capturing the workload on the server is straightforward if the server supports capture, but impacts the performance of the server. Capturing the workload on the network through port mirroring is almost as convenient as capture on the server, and given that most switches implement mirroring in hardware, has no impact on network or workload performance. Therefore, we have always chosen to capture the data through the use of port mirroring, if necessary, using multiple Ethernet ports for the mirrored packets.

The main challenge for raw packet capture is the underlying data rate. In order to parse NFS packets, we have to capture the complete packet. Because the capture host is not interacting with clients, it has no way to throttle incoming packets, so it needs to be able to capture at the full sustained rate or risk packet loss. To maximize flexibility, we want to write the data out to disk so that we can simplify the parsing and improve the error checking. This means that all of the incoming data eventually turns in to disk writes leading to the second challenge of maximizing effective disk space.

While the 1 second average rates may be low enough to fit onto the mirror ports, if the switch has insufficient buffering, packets can still be dropped. We discovered this problem on a switch that used per-port rather than per-card buffering. To eliminate the problem, we switched to 10Gbit mirror ports to reduce the need for switch-side buffering.

The capture host can also be overrun. At low data rates (900Mb/s, 70,000 packets/s), standard tcpdump on com-

modity hardware works fine. However, at high data rates (5Gb/s, 10^6 packets/s), traditional approaches are insufficient. Indeed, Leung [20] notes difficulties with packet loss using tcpdump on a 1 Gbit mirror port. We have developed three separate techniques for packet capture, all of which work better than tcpdump: *lindump* (user-kernel ring buffer), *driverdump* (in-kernel capture to files), and *endacedump* (hardware capture to memory).

3.1 Lindump

The Linux kernel includes a memory-mapped, shared ring buffer for packet capture. We modified the example lindump program to write out pcap files [8], the standard output format from tcpdump, and to be able to capture from more than one interface at the same time. We wrote the output files to an in-memory filesystem using mmap to reduce copies, and copied and compressed the files in parallel to disk. Using an HP DL580G2, a current 4 socket server circa 2003, lindump was able to capture about $3\times$ the packets per second (pps) as tcpdump and about $1.25\times$ the bandwidth. Combined with a somewhat higher burst rate while the kernel and network card buffered data, this approach was sufficient for mostly loss free captures at the animation company, and was the technique we used for all of the 2003 set of traces.

Packets are captured into files in tmpfs, an in-memory filesystem, and then compressed to maximize the effective disk space. If the capture host is mostly idle, we compressed with `gzip -9`. As the backlog of pending files increased, we reduced the compression algorithm to `gzip -6`, then to `gzip -1`, and finally to nothing. In practice this approach increased the effective disk size by $1.5\text{--}2.5\times$ in our experience as the data was somewhat compressible, but at higher input rates we had to fall back to reduced compression.

3.2 Driverdump

At another site, our 1Gbit lindump approach was insufficient because of packet bursts and limited buffering on the switch. Replacing the dual 1Gbit cards with a 10Gb/s card merely moved the bottleneck to the host and the packets were dropped on the NIC before they could be consumed by the kernel.

To fix this problem, we modified the network driver so that instead of passing packets up the network stack, it would just copy the packets in pcap format to a file, and immediately return the packet buffer to the NIC. A user space program prepared files for capture, and closed the files on completion. We called our solution *driverdump* since it performed all of the packet dumping in the driver.

Because driverdump avoids the kernel IP stack, it can capture packets faster than the IP stack could drop them. We increased the sustained packets per second over lindump by $2.25\times$ to 676,000pps, and sustained bandwidth by $1.5\times$ to 170MiB/s (note 1 MiB/s = 2^{20} bytes/s). We could handle short bursts up to 900,000 pps, and 215 MiB/s. This gave us nearly lossless capture to memory at the second site. Since the files were written into tmpfs, we re-used our technology for compressing and copying the files out to disk.

3.3 Endacedump

In 2007, we returned to the animation company to collect new traces on their faster NFS servers and 10Gb/s network. While an update of driverdump might have been sufficient, we decided to also try the Endace DAG 8.2X capture card [14]. This card copies and timestamps packets from a 10Gb/s network directly into memory. As a result, it can capture minimal size packets at full bandwidth, and is intended for doing in-memory analysis of networks. Our challenge was to get the capture out to disk, which was not believed to be feasible by our technical contacts at Endace.

To solve this problem, we integrated our adaptive compression technique into a specialized capture program, and added the lzf [21] compression algorithm, that compresses at about 100MiB/s. We also upgraded our hardware to an HP DL585g2 with 4 dual-core 2.8Ghz Opterons, and 6 14 disk SCSI trays. Our compression techniques turned our 20TiB of disk space into 30TiB of effective disk space. We experienced a very small number of packet drops because our capture card limited a single stream to PCI-X bandwidth (8Gbps), and required partitioning into two streams to capture 10Gb/s. Newer cards capture 10Gb/s in a single stream.

3.4 Discussion

Our capture techniques are directly applicable to anyone attempting to capture data from a networked storage service such as NFS, CIFS, or iSCSI. The techniques present a tradeoff. The simplest technique (lindump), is a drop in replacement for using tcpdump for full packet capture, and combined with our adaptive compression algorithm allows capture at over twice the rate of native tcpdump and expands the effective size of the disks by $1.5\times$. The intermediate technique increases the capture rates by an additional factor of $2\text{--}3\times$, but requires modification of the in-kernel network driver. Our most advanced techniques are capable of lossless full-packet capture at 10Gb/s, but requires purchasing special capture hardware.

Both the lindump and driverdump code are available in our source distribution [9]. These tools and techniques should eliminate problems of packet drops for capturing storage traces. Further details and experiments with the first two techniques can be found in [1].

4 Conversion from raw format

Once the data is captured, the second problem is parsing and converting that data to a easily usable format. The raw packet format contains a large amount of unnecessary data, and would require repeated, expensive parsing to be used for NFS analysis. There are four main challenges in conversion: representation, storage, performance and anonymization. *Data representation* is the challenge of deciding the logical structure of the converted data. *Storage format* is the challenge of picking a suitable physical structure for the converted data. *Conversion performance* is the challenge of making the conversion run quickly, ideally faster than the capture stage. *Trace anonymization* is the challenge of hiding sensitive information present in the data and is necessary for being able to release traces.

One lesson we learned after conversion is that the converter's version number should be included in the trace. As with most programs, there can be bugs. Having the version number in the trace makes it easy to determine which flaws need to be handled. For systems such as subversion or git, we recommend the atomic check-in ID as a suitable version number.

A second lesson was preservation of data. An NFS parser will discard data both for space reasons and for anonymization. Keeping underlying information, such as per packet conversion in addition to per NFS-request conversion can enable cross checking between analysis. We caught an early bug in our converter that failed to record packet fragments by comparing the packet rates and the NFS rates.

4.1 Data representation

One option for the representation is the format used in the Ellard [11] traces: one line per request or reply in a text file with field names to identify the different parameters in the RPC. This format is slow to parse, and works poorly for representing readdir, which has an arbitrary number of response fields. Therefore, we chose to use a more relational data structuring [7].

We have a primary data table with the common fields present in every request or reply, and an identifier for each RPC. We then have secondary tables that contain request-type specific information, such as a single table for RPC's

that include attributes, and a single table for read and write information. We then join the common table to the other tables when we want to perform an analysis that uses information in both. Because of this structure, a single RPC request or reply will have a single entry in the common table. However, a request/reply pair will have zero (no entry in the read/write table unless the operation is a read/write) or more entries (multiple attribute entries for readdir+) in other tables.

The relational structuring improves flexibility, and avoids reading unnecessary data for analyses that only need a subset of the data. For example, an analysis only looking at operation latency can simply scan the common table.

4.2 Storage format

Having decided to use a relational structuring for our data, we next needed to decide how to physically store the data. Three options were available to us: text, SQL, and DataSeries, our custom binary format [2] for storing trace data. Text is a traditional way of storing trace data, however, we were concerned that a text representation would be too large and too slow. Having later converted the Ellard traces to our format, we found that the analysis distributed with the traces used $25\times$ less CPU time when the traces and analysis used DataSeries, and ran $100\times$ faster on a 4 core machine. This disparity confirmed our intuition that text is a poor format for trace data.

SQL databases support a relational structure. However, the lack of extensive compression means that our datasets would consume a huge amount of space. We also expected that many complex queries would not benefit from SQL and would require extracting the entire tables through the slow SQL connection.

Therefore, we selected DataSeries as an efficient and compact format for storing traces. It uses a relational data model, so there are rows of data, with each row comprised of the same typed columns. A column can be nullable, in which case there is a hidden boolean field for storing whether the value is null. Groups of rows are compressed as a unit. Prior to compression, various transforms are applied to reduce the size of the data. First, duplicate strings are collapsed down to a single string. Second, values are delta compressed relative to either the same value in the previous row or another value in the same row. For example, the packet time values are delta compressed, making them more compressible by a general purpose compression algorithm.

DataSeries is designed for efficient access. Values are packed so that once a group of rows is read in, an anal-

ysis can iterate over them simply by increasing a single counter, as with a C++ vector. Individual values are accessed by an offset from that counter and a C++ cast. Byte swapping is automatically performed if necessary. The offset is not fixed, so the same analysis can read different versions of the data, provided the meaning of the fields has not changed. Efficient access to subsets of the data is supported by an automatically generated index.

DataSeries is designed for generality. It supports versioning on the table types so that an analysis can properly interpret data that may have changed in meaning. It has special support for time fields so that analysis can convert to and from different raw formats.

DataSeries is designed for integrity. It has internal checksums on both the compressed and the uncompressed data to validate that the data has been processed appropriately. Additional details on the format, additional transforms, and comparisons to a wide variety of alternatives can be found in the technical report [10].

4.3 Conversion performance

To perform the conversion in parallel, we divide the collected files into groups and process each group separately. We make two passes through the data. First, we parse the data and count the number of requests or replies. Second, we use those counts to determine the first record-id for each group, and convert the files. Since NFS parsing requires the request to parse the reply, we currently do not parse any request-reply pairs that cross a group boundary. Similarly, we do not do full TCP reconstruction, so for NFS over TCP, we parse multiple requests or replies if the first one starts at the beginning of the packet. These limitations are similar to earlier work, so we found them acceptable. We run the conversion locally on the 8-way tracing machine rather than a cluster because conversion runs faster than the 1Gbit LAN connection we had at the customer site (the tracing card does not act as a normal NIC). Conversion of a full data set (30TiB) takes about 3 days.

We do offline conversion from trace files, rather than online conversion, primarily for simplicity. However, a side benefit was that our converter could be paranoid and conservative, rather than have it try to recover from conversion problems, since we could fix the converter when it was mis-parsing or was too conservative. The next time we trace, we plan to do more on-the-fly conversion by converting early groups and deleting those trace files during capture so that we can capture longer traces.

4.4 Trace anonymization

In order to release the traces, we have to obscure private data such as filenames. There are three primary ways to map values in order to anonymize them:

1. **unique integers.** This option results in the most compact identifiers (≤ 8 bytes), but is difficult to calculate in parallel and requires a large translation table to maintain persistent mappings and to convert back to the original data.
2. **hash/HMAC.** This option results in larger identifiers (16-20 bytes), but enables parallel conversion. A keyed HMAC [5] instead of a hash protects against dictionary attacks. Reversing this mapping requires preserving a large translation table.
3. **encrypted values.** This option results in the longest identifiers since the encrypted value will be at least as large as the original value. It is parallizable and easily reversible provided the small keys are maintained.

We chose the last approach because it preserved the maximum flexibility, and allowed us to easily have discussions with the customer about unexpected issues such as writes to what should have been a read-only filesystem. Our encryption includes a self-check, so we can convert back to real filenames by decrypting all hexadecimal strings and keeping the ones that validate. We have also used the reversibility to verify for a colleague that they properly identified the ‘.’ and ‘..’ filenames.

We chose to encrypt entire filenames since the suffixes are specific to the animation process and are unlikely to be useful to people. This choice also simplified the discussions about publishing the traces. Since we can decrypt, we could in the future change this decision.

The remaining values were semi-random (IP addresses in the 10.* network, filehandles selected by the NFS servers), so we pass those values through unchanged. We decided that the filehandle content, which includes for our NFS servers the filesystem containing the file, could be useful for analysis. Filehandles could also be anonymized.

All jobs in the customers’ cluster were being run as a common user, so we did not capture user identifiers. Since they are transitioning away from that model, future traces would include unchanged user identifiers and group identifiers. If there were public values in the traces, then we would have had to apply more sophisticated anonymization [27].

5 Analysis techniques

Analyzing the very large amount of data that we collected required us to adopt and develop new analysis techniques. The most important property that we aimed for was bounded memory, which meant that we needed to have streaming analysis. The second property that we wanted was efficiency, because without compute-time efficiency, we would not be able to analyze complete datasets. One of our lessons is that these techniques allowed us to handle the much larger datasets that we have collected.

5.1 Approximate quantiles

Quantiles are better than simple statistics or histograms because they do not accidentally combine separate measurements regardless of distribution. Unfortunately, for our data, calculating exact quantiles is impractical. For a single dataset, we collect multiple statistics with a total of about 200 billion values. Storing all these values would require ≈ 1.5 TiB of memory, which makes it impractical for us to calculate exact quantiles.

However, there is an algorithm from the database field for calculating approximate quantiles in bounded memory [22]. A q -quantile of a set of n data elements is the element at position $\lceil q * n \rceil$ in the sorted list of elements indexed from 1 to n . For approximate quantiles, the user specifies two numbers ϵ , the maximum error, and N , the maximum number of elements. Then when the program calculates quantile q , it actually gets a quantile in the range $[q - \epsilon, q + \epsilon]$.

Provided that the total number of elements is less than N , the bound is guaranteed. We have found that usually the error is about $10\times$ better than specified. For our epsilon of 0.005 (sufficient to guarantee all percentiles are distinct), instead of needing ≈ 1.5 TiB, we only need ≈ 1.2 MiB, an improvement of $> 10^6$. This dramatic improvement means we can run the analysis on one machine, and hence process multiple sets in parallel. The performance cost of the algorithm is about the same as sorting since the algorithm does similar sorting of subsets and merging of subsets. Details on how the algorithm works can be found in [22] or our software distribution.

5.2 Data cube

Calculating aggregate or roll-up statistics is an important part of analyzing a workload. For example, consider the information in the common NFS table: $\langle \text{time, operation, client-id, and server-id} \rangle$. We may want to calculate the total number of operations performed by client 5, in which

case we want to count the number of rows that match $\langle *, *, 5, * \rangle$.

The cube [16] is a generalization of the group-by operations described above. Given a collection of rows, it calculates the set of unique values for each column $U(c)$, adds the special value 'ANY' to the set, and then generates one row for each member of the cross-product $U(1) \times U(2) \times \dots U(n)$.

We implemented an efficient templated version of the cube operator for use in data analysis. We added three features to deal with memory usage. First, our cube can only include rows with actual values in it. This eliminates the large number of rows from the cross-product that match no rows in the base data. Second, we can further restrict which rows are generated. For example, we have a large number of client id's, and so we can avoid cubing over entries with both the client and operation specified to reduce the number of statistics calculated. Third, we added the ability to prune values out of the cube. For example, we can output cube values for earlier time values and remove them from the data structure once we reach later time values since we know the data is sorted by time.

The cube allows us to easily calculate a wide variety of summary statistics. We had previously manually implemented some of the summary statistics by doing explicit roll-ups for some of the aggregates described in the example. We discovered that the general implementation was actually more efficient than our manual one because it used a single hash table for all of the data rather than nested data structures, and because we tuned the hash function over the tuple of values to be calculated efficiently.

5.3 HashTable

Our hash-table implementation [9] is a straightforward chained-hashing implementation. In our experiments it is strictly better in both performance and memory than the GNU C++ hash table. It uses somewhat more memory than the Google sparse hash [15], but performs almost as well as the dense hash; it is strictly faster than the g++ STL hash. We added three unusual features. First, it can calculate its memory usage, allowing us to determine what needs to be optimized. Second, it can partially reset iterators, which allows for safe mutating operations on the hash table during iteration, such as deleting a subset of the values. Third, it can return the underlying hash chains, allowing for sorting the hash table without copying the values out. This operation destroys the hash table, but the sort is usually done immediately before deleting the table, and reduces memory usage by $2\times$.

5.4 Rotating hash-map

Limiting memory usage for hash tables where the entries have unknown lifespan presents some challenges. Consider the sequentiality metric: so long as accesses are active to the file, we want to continue to update the run information. Once the file becomes inactive for long enough, we want to calculate summary statistics and remove the general statistics from memory. We could keep the values in an LRU data-structure. However if our analysis only needs a file id and last offset, then the forward and backwards pointers for LRU would double the memory usage. A clock-style algorithm would require regular full scans of the entire data structure.

We instead solve this problem by keeping two hash-maps, the *recent* and *old* hash-maps. Any time a value is accessed, it is moved to the recent hash-map if it is not already there. At intervals, the program will call the `rotate(fn)` operation which will apply `fn` to all of the (key,value) pairs in the old hash map, delete that map, assign the recent map to the old map and create a new recent map.

Therefore, if the analysis wants to guarantee any gap of up to 60 seconds will be considered part of the same run, it just needs to call `rotate()` every 60 seconds. Any value accessed in the last 60 seconds will remain present in the hash-map. We could reduce the memory overhead somewhat by keeping more than two hash-maps at the cost of additional lookups, but we have so far found that the rotating hash-map provides a good tradeoff between minimizing memory usage and maximizing performance. We believe that the LRU approach would be more effective if the size of the data stored in the hash map were larger, and the hash-map could compact itself so that scattered data entries do not consume excess space.

5.5 Graphing with mercury-plot

Once we have summarized the data from `DataSet` using the techniques described above, we need to graph and subset the data. We combined SQL, Perl, and gnuplot into a tool we call `mercury-plot`. SQL enables sub-setting and combining data. For example if we have data on 60 second intervals, it is easy to calculate min/mean/max for 3600 second intervals, or with the cube to select out the subset of the data that we want to use. We use Perl to handle operations that the database can not handle. For example, in the cube, we represent the 'ANY' value as null, but SQL requires a different syntax to select for null vs. a specific value. We hide this difference in the Perl functions. In practice, this allows us to write very simple commands such as `plot quantile as`

x, value as y from nfs_hostinfo_cube where operation = 'read' and direction = 'send' to generate a portion of the graph. This tool allows us to deal with the millions of rows of output that can come from some of the analysis. To ease injection of data from the C++ DataSeries analysis, one lesson we learned is analysis should have a mode that generates SQL insert statements in addition to human readable output.

6 Analysis

Analyzing very large traces can take a long time. While our custom binary format enables efficient analysis, and our analysis techniques are efficient, it can still take 4-8 hours to analyze a single set of the 2007 traces. In practice, we analyze the traces in parallel on a small cluster of four core 2.4GHz Opterons. Our analysis typically becomes bottle-necked on the file servers that serve up to 200MiB/s each once an analysis is running on more than 20 machines.

We collected data at two times: August 2003 - February 2004 (anim-2003), and January 2007 - October 2007 (anim-2007). We collected data using a variety of mirror ports within the company's network. The network design is straightforward: there is a redundant set of core routers, an optional mid-tier of switches to increase the effective port count of the core, and then a collection of edge switches that each cover one or two racks of rendering machines. Most of our traces were taken by mirroring links between rendering machines and the rest of the network. For each collected dataset, we would start the collection process, and let it run either until we ran out of disk space, or we had collected all the data we wanted. Each of these runs comprises a set. We have 21 sets from 2003, and 8 sets from 2007.

We selected a subset of the data to present, two datasets from 2003 and two from 2007. The sets were selected both because they are representative of the more intensive traces from both years, and to show some variety in the data. We identified clients as hosts that sent requests, servers as hosts that sent replies, and caches as hosts that acted as both clients and servers. Further information on each dataset can be found on the trace download page [4].

- anim-2003/set-5: A trace of 79 clients accessing 50 NFS servers. NFS caches are seen as servers in this trace.
- anim-2003/set-12: A trace of 1634 clients accessing 1 NFS server. NFS caches are seen as clients in this trace.

- anim-2007/set-2: A trace of 273 clients accessing 40 NFS servers at the same site as anim-2003/set-5. The other traces of clients at this site are similar to set-2.
- anim-2007/set-5: A trace of 135 clients accessing 50 NFS servers, and 8 caches acting as both clients and servers, although because of the port mirroring setup, we did not see some of the responses from the caches. This trace is at a different site from set-2 and shows higher burstiness.

6.1 Capture performance

We start our analysis by looking at the performance of our capture tool. This validates our claims that we can capture packets at very high data rates. We examine the capture rate of the tool by calculating the megabits/s (Mbps) and kilo-packets/s (kpps) for 20 overlapping sub-intervals of a specified length. For example if our interval length is 60 seconds, then we will calculate the bandwidth for the interval 0s-60s, 3s-63s, 6s-66s, ... end-of-trace. We chose to calculate the bandwidth for overlapping intervals so that we would not incorrectly measure the peaks and valleys of cyclic patterns aligned to the interval length. We use the approximate quantile so we can summarize results with billions of underlying data points. For example, we have 11.6 billion measurements for anim-2007/set-0 at a 1ms interval length. This corresponds to the 6.7 days of that trace.

Figure 2(a) shows anim-2007/set-5 at different interval lengths. This graph shows the effectiveness of our tracing technology, as we have sustained intervals above 3Gb/s (358MiB/s), and 1ms intervals above 4Gb/s (476MiB/s). Indeed these traces show the requirement for high speed tracing, as 5-20% of the trace intervals have sustained intervals above 1Gbit, which is above the rate at which Leung [20] noted their tracing tool started to drop packets. The other sets from anim-2007 are somewhat less bursty, and the anim-2003 data shows much lower peaks because of our more limited tracing tools, and a wider variety of shapes, because we traced at more points in the network.

Figure 2(a) also emphasizes how bursty the traffic was during this trace. While 50% of the intervals were above 500Mbit/s for 60s intervals, only 30% of the intervals were above 500Mbit/s for 1ms intervals. This burstiness is expected given that general Ethernet and filesystem traffic have been shown to be self-similar [17, 19], which implies the network traffic is also bursty. It does make it clear that we need to look at short time intervals in order to get an accurate view of the data.

Figure 2(b) shows the tail of the distributions for the capture rates for two of the trace sets. The relative sim-

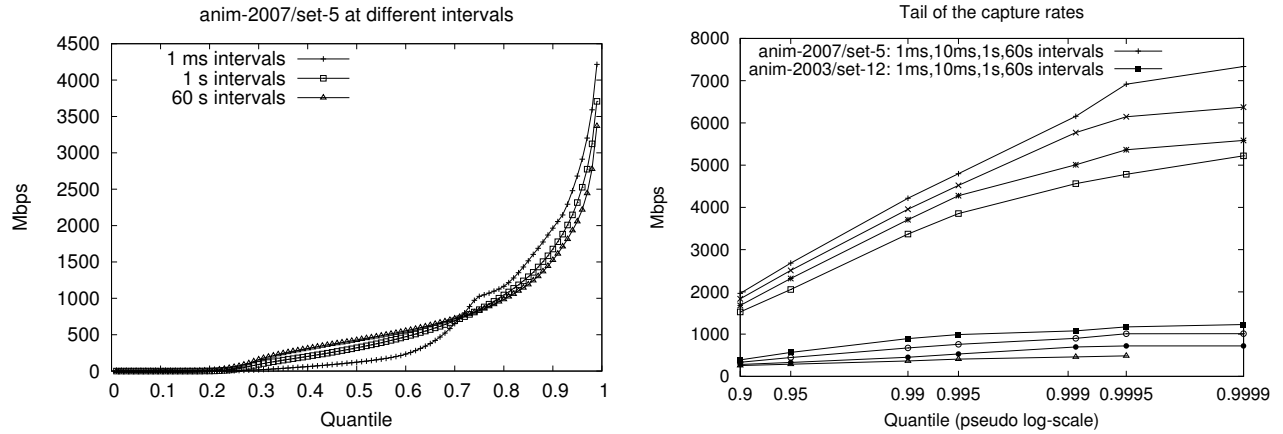


Figure 2: Bandwidth measured in the collection process. In Figure (b), anim-2007/set-5 at different intervals is the top group of 4 lines, and anim-2003/set-12 is the bottom group of 4 lines. With 60s intervals, anim-2003/set-12 does not show the 0.9999 quantile because there were insufficient data points.

operation	anim-2003/set-12		anim-2003/set-5		anim-2007/set-2		anim-2007/set-5	
	Mops	bytes/op	Mops	bytes/op	Mops	bytes/op	Mops	bytes/op
readdir	4.579	281	1.132	3940	28.318	4089	18.350	4071
readdirplus	0.632	2307	0.000	n/a	32.806	1890	20.271	2001
readlink	0.081	74	0.049	79	25.421	204	42.335	203
fsstat	19.875	56	50.416	56	0.017	180	0.003	180
write	14.546	9637	30.236	7880	32.390	13562	45.177	15015
lookup	134.108	83	82.823	92	643.854	239	807.127	235
read	345.743	1231	165.969	7855	1460.669	14658	1761.199	12301
access	1.858	136	0.000	136	4000.204	136	3570.404	136
getattr	244.650	104	967.961	104	6598.515	124	2756.785	123
total	768.053	790	1301.364	1274	12851.102	1833	9034.968	2599

Table 1: symlink, rmdir, mkdir, and rename were pruned as there were fewer than 1 million operations; fsinfo, link, null, create, remove, and setattr were pruned as there were fewer than 10 million operations. The Mops column could be calculated from nfsstat, but the bytes/op column could not.

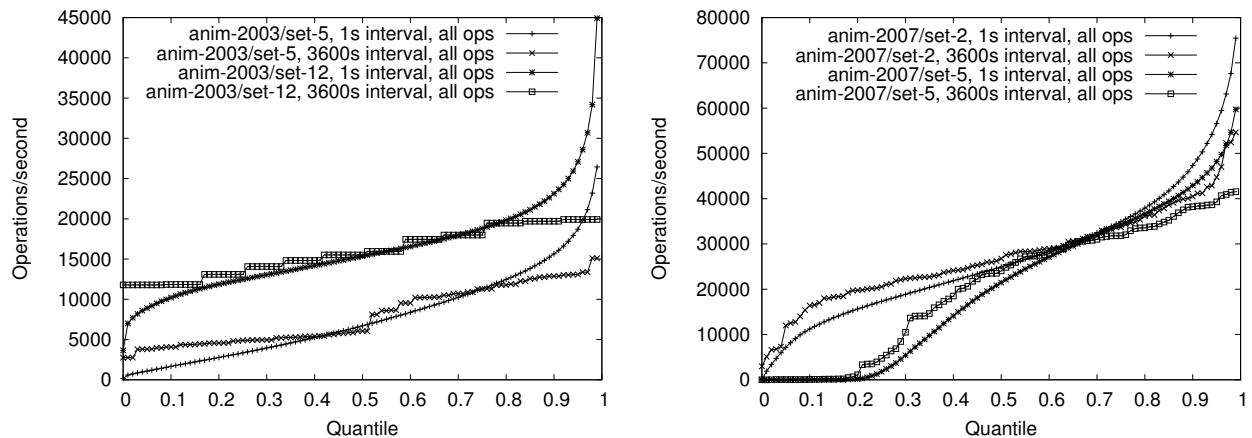


Figure 3: Operation rates, as quantiles, for anim-2003, anim-2007.

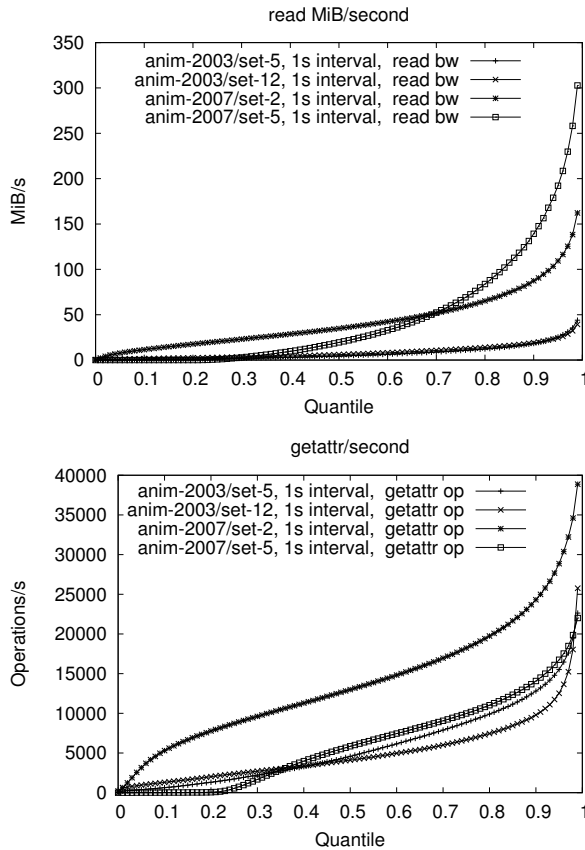


Figure 4: Bandwidth for reads and operation rate for getattrs in the four traces.

ilarity between the Mbps and kpps graphs is simply because packet size distributions are relatively constant. The traces show the remarkably high burstiness of the 2007 traces. While 90% of the 1ms intervals are below 2Gb/s, 0.1% are above 6Gb/s. We expect we would have seen slightly higher rates, but because of our configuration error for the 2007 capture tool, we could not capture above about 8Gb/s.

6.2 Basic NFS analysis

Examining the overall set of operations used by a workload provides insight into what operations need to be optimized to support the workload. Examining the distribution of rates for the workload tells us if the workload is bursty, and hence we need to handle a higher rate than would be implied by mean arrival rates, and if there are periods of idleness that could be exploited.

Table 1 provides an overview of all the operations that

dataset	1s ops/s	3600s ops/s	ratio
anim-2003/set-5	26,445	15,110	1.75×
anim-2003/set-12	44,926	19,923	2.25×
anim-2007/set-2	75,457	54,657	1.38×
anim-2007/set-5	59,727	41,550	1.44×

Table 2: Operation rate ratios

occurred in the four traces we are examining in more detail. It shows a number of substantial changes in the workload presented to the NFS subsystem. First, the read and write sizes have almost doubled from the anim-2003 to anim-2007 datasets. This trend is expected, because the company moved from NFSv2 to NFSv3 between the two tracing periods, and set the v3 read/write size to 16KiB. The company told us they set it to that size based on performance measurements of sequential I/O. The NFS version switch also accounts for the increase in access calls (new in v3), and readdirplus (also new in v3).

We also see that this workload is incredibly read-heavy. This is expected; the animation workload reads a very large number of textures, models, etc. to produce a relatively small output frame. However, we believe that our traces under-estimate the number of write operations. We discuss the write operation underestimation below. The abnormally low read size for set-12 occurred because that server was handling a large number of stale filehandle requests. The replies were therefore small and pulled down the bytes/operation. We see a lot more getattr operations in set-5 than set-12 because set-12 is a server behind several NFS-caches, whereas set-5 is the workload before the NFS-caches.

Table 2 and Figures 3(a,b) show how long averaging intervals can distort the load placed on the storage system. If we were to develop a storage system for the hourly loads reported in most papers, we would fail to support the substantially higher near peak (99%) loads seen in the data. It also hides periods of idleness that could be used for incremental scrubbing and data reorganization. We do not include the traditional graph of ops/s vs. time because our workload does not show a strong daily cycle. Animators submit large batches of jobs in the evening that keep the cluster busy until morning, and keep the cluster busy during the day submitting additional jobs. Since the jobs are very similar, we see no traditional diurnal pattern in the NFS load, although we do see the load go to zero by the end of the weekend.

Figure 4 shows the read operation MiB/s and the getattr operations/s. It shows that relative to the amount of data being transferred, the number of getattrs has been reduced, likely a result of the transition from NFSv2 to

NFSv3. The graph shows the payload data transferred, so it includes the offset and filehandle of the read request, and the size and data in the reply, but does not include IP headers or NFS RPC headers. It shows that the NFS system is driven heavily, but not excessively. The write operations/s graph (not shown for space reasons) implies that the write bandwidth has gotten more bursty, but has stayed roughly constant.

This result led us to further analyze the data. We were surprised that write bandwidth did not increase, even though it is not implausible, as the frame output size has not increased. We analyzed the traces to look for missing operations in the sequence of transaction ids, automatically inferring if the client is using a big-endian or little-endian counter. The initial results looked quite good: anim-2007/set-2 showed 99.7% of the operations were in sequence, anim-2007/set-5 showed 98.4%, and counting the skips of 128 transactions or less, we found only 0.21% and 0.50% respectively (the remaining entries were duplicates or ones that we could not positively tell if they were in sequence or a skip). However, when we looked one level deeper at the operation that preceded a skip in the sequence, we found that 95% of the skips followed a write operation for set-2, and 45% for set-5. The skips in set-2 could increase the write workload by a factor of $1.5\times$ if all missing skips after writes are associated with writes. We expected a fair number of skips for set-5 since we experienced packet loss under load, but we did not expect it for set-2.

Further examination indicated that the problem came about because we followed the same parsing technique for TCP packets as was used in `nfsdump2` [11]. We started at the beginning of the packet and parsed all of the RPCs that we found that matched all required bits to be RPCs. Unfortunately, over TCP, two back to back writes will not align the second write RPC with the packet header, and we will miss subsequent operations until they re-align with the packet start. While the fraction of missing operations is small, they are biased toward writes requests and read replies. Since we had saved IP-level trace information as well as NFS-level, we could write an analysis that conservatively calculated the bytes of IP packets that were not associated with an NFS request or reply. Counting a direction of a connection if it transfers over 10^6 bytes, we found for anim-2007/set-2 that we can account for >90% of the bytes for 87% of the connections, and for anim-2007/set-5 that we can account for >90% of the bytes for >70% of the connections. The greater preponderance of missing bytes relative to missing operations reinforces our analysis above that the losses are due to non-aligned RPC's since we are missing very few op-

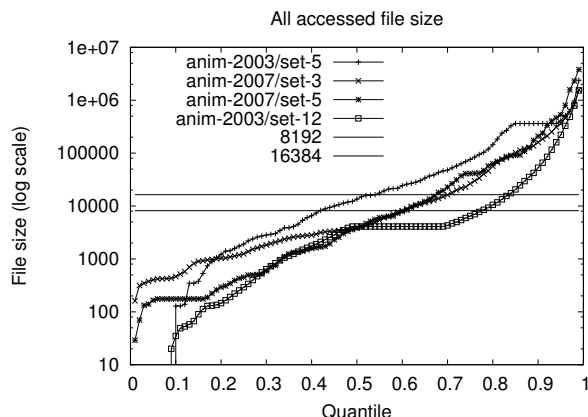


Figure 5: File size distribution for all accessed files.

erations, but many more bytes, and reads and writes have a high byte to operation ratio.

While this supports our lesson that retaining lower level information is valuable, this analysis also leads us to another one of our lessons: extensive validation of the conversion tool is important. Both validation through validation statistics, and through the use of a known workload that exercises the capture tools. An NFS replay tool [32] could be used to generate a workload, the replayed workload could be captured, and the capture could be compared to the original replayed workload. This comparison has been done to validate a block based replay tool [3], but has not been done to validate an NFS tracing tool, as the work has simply assumed tracing was correct. We believe a similar flaw is present in earlier traces [11] because the same parsing technique was used, although we do not know how much those traces were affected.

6.3 File sizes

File sizes affect the potential internal fragmentation for a filesystem. They affect the maximum size of I/Os that can be executed, and they affect the potential sequentiality in a workload.

Figure 5 shows the size of files accessed in our traces. It shows that most files are small enough to be read in a single I/O: 40-80% of the files are smaller than 8KiB (NFSv2 read size) for the 2003 traces, and 70% of the files are smaller than 16KiB for the 2007 traces. While there are larger files in the traces, 99% of the files are smaller than 10MiB. The small file sizes present in this workload, and the preponderance of reads suggest that a flash file system [18] or MEMS file system [29] could support a substantial portion of the workload.

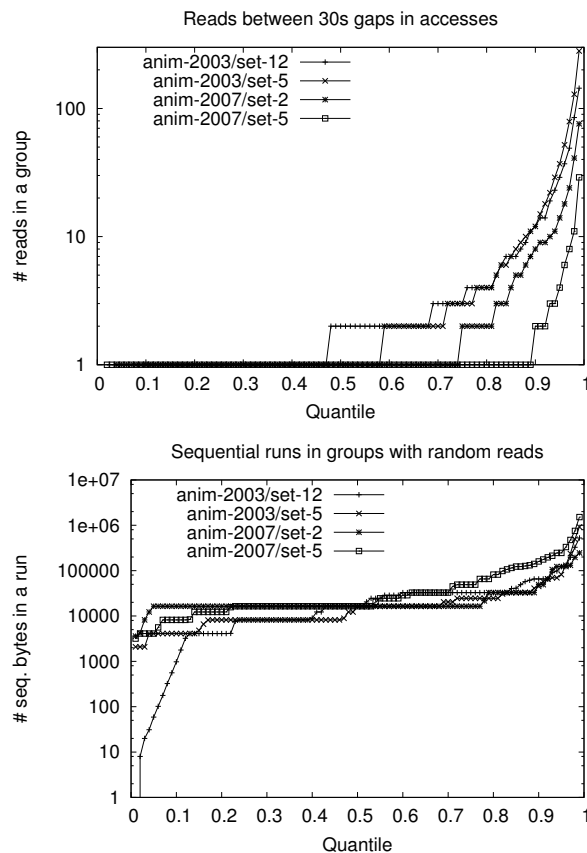


Figure 6: Number of reads or sequential bytes in a single group (more than 30s gap between I/Os);

6.4 Sequentiality

Sequentiality is one of the most important properties for storage systems because disks are much more efficient when handling sequential data accesses. Prior work has presented various methods for calculating sequentiality. Both Ellard [12] and Leung [20] split accesses into groups and calculate the sequentiality within the group. Ellard emulates opens and closes by looking for 30s groups in the access pattern. Ellard tolerates small gaps in the request stream as sequential, e.g. an I/O of 7KiB at offset 0 followed by an I/O of 8KiB at offset 8KiB would be considered sequential.

Ellard also reorders I/Os to deal with client-side reordering. In particular, Ellard looks forward a constant amount from the request time to find I/Os that could make the access pattern more sequential. This constant was determined empirically. Leung treats the first I/O after an open as sequential, essentially assuming that the server will prefetch the first few bytes in the file or that the file

is contiguous with the directory entry as with immediate files [24]. For NFS, the server may not see a lookup before a read, depending on whether the client has used `readdir+` to get the filehandle instead of a lookup.

We determine sequentiality by reordering within temporally overlapping requests. Given two I/Os, A and B, if the request-reply intervals overlap, then we are willing to reorder the requests to improve estimated sequentiality. We believe this is a better model because the NFS server could reorder those I/Os. In practice, Figure 7 shows that for our traces this reordering makes little difference. Allowing reordering an additional 10ms beyond the reply of I/O A slightly increases the sequentiality, but generally not much more than just for overlapping requests.

We also decide on whether the first I/O is sequential or random based on additional I/Os. If the second I/O (after any reordering) is sequential to the first one, then the first I/O is sequential, otherwise it is random. If there is only one I/O to a particular file, then we consider the I/O to be random since the NFS server would have to reposition to that file to start the read.

Given our small file sizes, it turns out that most accesses count as random because they read the entire file in a single I/O. We can see this in Figure 6(a), which shows the number of reads in a group. Most groups are single I/O groups (70-90% in the 2007 traces). We see about twice as many I/Os in the 2003 traces, because the I/Os in the 2003 traces are only 8KiB, rather than 16KiB.

Sequential runs within a random group are more interesting. Figure 6(b) shows the number of bytes accessed in sequential runs within a random group. We can see that if we start accessing a file at random, most (50-80%) of the time we will do single or double I/O accesses (8-32KiB). However we also get some extended runs within a random group, although 99% of the runs are less than 1MiB.

7 Conclusions

We have described three improved techniques for packet capture on networks. The easily adopted technique should allow anyone capturing NFS, CIFS, or iSCSI traffic from moderate performance storage systems (≤ 1 Gbit) to capture traffic with no losses. The most advanced technique allows lossless capture for 5-10Gbit storage systems, which is at the high end of most file storage systems. The primary lesson from this part of the work is that lossless 1Gbit packet capture is straightforward and up to 10Gbit is possible with an investment in development time or specialized hardware.

We have provided guidelines for conversion for future practitioners: parallelizing the conversion, retaining

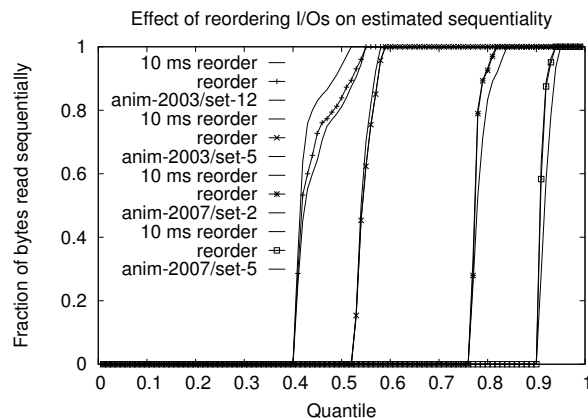


Figure 7: Each line group shows host estimated sequentiality was affected by allowing, in order: reordering of I/Os within 10ms of the reply, reordering within the request-reply window, or no reordering. The small horizontal change shows that reordering this workload has a negligible effect on sequentiality.

lower-level information, using reversible anonymization, approaches for testing the conversion tools, and tagging the trace data with version information.

We have described our binary storage format, which uses chunked compression with multiple possible compression techniques, typed relational-style data structuring, delta encoding, and type-safe, high-speed accessors. It improves over prior storage formats by up to 100 \times .

We have described our techniques for improved memory and performance efficiency to enable analysis of very large data sets. We explained the cube and approximate quantile techniques that we adopted from the database literature, and our hashtable, rotating hash-map, and plotting techniques that we use for analyzing the data.

We have analyzed our NFS workload examining some of the different properties found in a feature animation workload and demonstrating that our techniques are effective. We found that our workload had much more activity than previously described workloads, and that the file size and sequentiality is different than those workloads.

The tools described in this paper are available as open source from <http://tesla.hpl.hp.com/opensource/>, and the traces are available from <http://apotheca.hpl.hp.com/pub/datasets/animation-bear/>.

8 Acknowledgements

The author would like to thank Alistair Veitch, Jay Wylie, Kimberly Keeton, our shepherd Daniel Ellard and the anonymous reviewers for their comments that have greatly improved our paper.

References

- [1] Eric Anderson and Martin Arlitt. Full Packet Capture and Offline Analysis on 1 and 10 Gb/s Networks. Technical Report 156, HP Labs, 2006. <http://www.hpl.hp.com/techreports/2006/HPL-2006-156.html>, accessed January 2009.
- [2] Eric Anderson, Martin Arlitt, Charles B. Morrey III, and Alistair Veitch. DataSeries: An Efficient, Flexible Data Format for Structured Serial Data. *Operating Systems Review*, 43(1):70–75, 2009.
- [3] Eric Anderson, Mahesh Kallahalla, Mustafa Uysal, and Ram Swaminathan. Buttress: A Toolkit for Flexible and High Fidelity I/O Benchmarking. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 45–58, 2004.
- [4] Animation traces: <http://apotheca.hpl.hp.com/pub/datasets/animation-bear/>, accessed January 2009.
- [5] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology – Crypto 96*, pages 1–15, 1996. <http://www.cs.ucsd.edu/~mihir/papers/kmd5.pdf>, accessed January 2009.
- [6] Brent Callaghan, Brian Pawlowski, and Peter Staubach. NFS Version 3 Protocol Specification, RFC 1813, June 1995. <http://www.ietf.org/rfc/rfc1813.txt>, accessed January 2009.
- [7] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. In *Communications of the ACM*, volume 13, pages 377–387, 1970.
- [8] Loris Degioanni, Fulvio Risso, and Gianluca Varenni. PCAP Next Generation Dump File Format, March 2004. <http://www.winpcap.org/ntar/draft/PCAP-DumpFileFormat.html>, accessed January 2009.
- [9] DataSeries open source software: <http://tesla.hpl.hp.com/opensource/>, accessed January 2009.
- [10] <http://tesla.hpl.hp.com/opensource/>

- DataSeries-tr-snapshot.pdf, accessed January 2009.
- [11] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. <http://iotta.snia.org/traces/list/NFS>, accessed January 2009.
 - [12] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST 2003)*, pages 203–216, 2003.
 - [13] Daniel Ellard and Margo Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proceedings of the 17th Large Installation System Administration Conference (LISA'03)*, pages 73–85, 2003.
 - [14] <http://www.endace.com/dag-network-monitoring-cards.html>, accessed January 2009.
 - [15] <http://goog-sparsehash.sourceforge.net/>, accessed January 2009.
 - [16] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In *Data Mining and Knowledge Discovery*, volume 1, pages 29–53, 1997. <ftp://ftp.research.microsoft.com/pub/tr/tr-97-32.doc>, accessed January 2009.
 - [17] Steven D. Gribble, Gurmeet Singh Manku, Drew Roselli, Eric A. Brewer, Timothy J. Gibson, and Ethan L. Miller. Self-similarity in file systems. *SIGMETRICS Performance Evaluation Review*, 26(1):141–150, 1998.
 - [18] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *Proceedings of the Winter 1995 USENIX Technical Conference*, pages 155–164, 1995.
 - [19] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the Self-Similar Nature of Ethernet Traffic (Extended Version). *IEEE/ACM Transactions on Networking*, 2:1–15, 1994.
 - [20] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 213–226, June 2008.
 - [21] lzf compression library: <http://www.goof.com/pgc/marc/liblzf.html>, accessed January 2009.
 - [22] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Special Interest Group on Management of Data (SIGMOD)*, pages 426–435, 1998.
 - [23] Microsoft. System call traces: <http://iotta.snia.org/traces/list/SystemCall>, accessed January 2009.
 - [24] Sape J. Mullender and Andrew S. Tanenbaum. Immediate Files. *Software – Practice and Experience*, 14(4):365–368, April 1984. <http://dare.uvu.vu.nl/bitstream/1871/2604/1/11033.pdf>, accessed January 2009.
 - [25] Bill Nowicki. NFS: Network File System Protocol Specification, RFC 1094, March 1989. <http://www.ietf.org/rfc/rfc1094.txt>, accessed January 2009.
 - [26] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. *Operating Systems Review*, 19(5):15–24, 1985.
 - [27] Ruoming Pang, Mark Allman, Vern Paxson, and Jason Lee. The devil and packet trace anonymization. *SIGCOMM Computer Communication Review*, 36(1):29–38, January 2006.
 - [28] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and David Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 137–152, 1994.
 - [29] Steven W. Schlosser and Gregory R. Ganger. MEMS-based Storage Devices and Standard Disk Interfaces: A Square Peg in a Round Hole? In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 87–100, 2004.
 - [30] <http://www.tcpdump.org/>, accessed January 2009.
 - [31] <http://www.wireshark.org/>, accessed January 2009.
 - [32] Ningning Zhu, Jiawu Chen, and Tzi-Cker Chiueh. TBBT: Scalable and Accurate Trace Replay for File Server Evaluation. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST 2005)*, pages 323–336, 2005.

Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems

Andrew W. Leung^{*} Minglong Shao[†] Timothy Bisson[†] Shankar Pasupathy[†] Ethan L. Miller^{*}

^{*}*University of California, Santa Cruz*

{aleung, elm}@cs.ucsc.edu

[†]*NetApp*

{minglong, tbisson, shankarp}@netapp.com

Abstract

The scale of today's storage systems has made it increasingly difficult to find and manage files. To address this, we have developed Spyglass, a file metadata search system that is specially designed for large-scale storage systems. Using an optimized design, guided by an analysis of real-world metadata traces and a user study, Spyglass allows fast, complex searches over file metadata to help users and administrators better understand and manage their files.

Spyglass achieves fast, scalable performance through the use of several novel metadata search techniques that exploit metadata search properties. Flexible index control is provided by an index partitioning mechanism that leverages namespace locality. Signature files are used to significantly reduce a query's search space, improving performance and scalability. Snapshot-based metadata collection allows incremental crawling of only modified files. A novel index versioning mechanism provides both fast index updates and "back-in-time" search of metadata. An evaluation of our Spyglass prototype using our real-world, large-scale metadata traces shows search performance that is 1-4 orders of magnitude faster than existing solutions. The Spyglass index can quickly be updated and typically requires less than 0.1% of disk space. Additionally, metadata collection is up to 10 \times faster than existing approaches.

1 Introduction

The rapidly growing amounts of data in today's storage systems makes finding and managing files extremely difficult. Storage users and administrators need to efficiently answer questions about the properties of the files being stored in order to properly manage this increasingly large sea of data. Metadata search, which involves indexing file metadata such as inode fields and extended attributes, can help answer many of these questions [26].

Metadata search allows point, range, top- k , and aggregation search over file properties, facilitating complex, ad hoc queries about the files being stored. For example, it can help an administrator answer "which files can be moved to second tier storage?" or "which application's and user's files are consuming the most space?". Metadata search can also help a user find his or her ten most recently accessed presentations or largest virtual machine images. Efficiently answering these questions can greatly improve how user and administrator manage files in large-scale storage systems.

Unfortunately, fast and efficient metadata search in large-scale storage systems is difficult to achieve. Both customer discussions [37] and personal experience have shown that existing enterprise search tools that provide metadata search [4, 14, 17, 21, 30] are often too expensive, slow, and cumbersome to be effective in large-scale systems. *Effective* metadata search must meet several requirements. First, it must be able to quickly gather metadata from the storage system. We have observed commercial systems that took 22 hours to crawl 500 GB and 10 days to crawl 10 TB. Second, search and update must be fast and scalable. Existing systems typically index metadata in a general-purpose DBMS. However, DBMSs are not a perfect fit for metadata search, which can limit their performance and scalability in large-scale systems. Third, resource requirements must be low. Existing tools require dedicated CPU, memory, and disk hardware, making them expensive and difficult to integrate into the storage system. Fourth, the search interface must be flexible and easy to use. Metadata search enables complex file searches that are difficult to ask with existing file system interfaces and query languages. Fifth, search results must be secure; many existing systems either ignore file ACLs or significantly degrade performance to enforce them.

To address these issues, we developed Spyglass, a novel metadata search system that exploits file metadata properties to enable fast, scalable search that can be em-

bedded within the storage system. To guide our design, we collected and analyzed file metadata snapshots from real-world storage systems at NetApp and conducted a survey of over 30 users and IT administrators. Our design introduces several new metadata search techniques. *Hierarchical partitioning* is a new method of namespace-based index partitioning that exploits namespace locality to provide flexible control of the index. *Signature files* are used to compactly describe a partition's contents, helping to route queries only to relevant partitions and prune the search space to improve performance and scalability. A new *snapshot-based* metadata collection method provides scalable collection by re-crawling only the files that have changed. Finally, *partition versioning*, a novel index versioning mechanism, enables fast update performance while allowing “back-in-time” search of past metadata. Spyglass does not currently address search interface or security, which are left to future work.

An evaluation of our Spyglass prototype, using our real-world, large-scale metadata traces, shows that search performance is improved 1–4 orders of magnitude compared to basic DBMS setups. Additionally, search performance is scalable; it is capable of searching hundreds of millions of files in less than a second. Index update performance is up to $40\times$ faster than basic DBMS setups and scales linearly with system size. The index itself typically requires less than 0.1% of total disk space. Index versioning allows “back-in-time” metadata search while adding only a tiny overhead to most queries. Finally, our snapshot-based metadata collection mechanism performs $10\times$ faster than a straw-man approach. Our evaluation demonstrates that Spyglass can leverage file metadata properties to improve how files are managed in large-scale storage systems.

This remainder of this paper is organized as follows. Section 2 provides additional metadata search motivation and background. Section 3 presents the Spyglass design. Our prototype is evaluated in Section 4. Related work is discussed in Section 5, with future work and conclusions in Section 6.

2 Background

This section describes and motivates the use of file metadata search and includes a discussion of real-world query and metadata characteristics.

2.1 File Metadata

File metadata, such as inode fields (*e.g.*, size, owner, timestamps, *etc.*), generated by the storage system and extended attributes (*e.g.*, document title, retention policy, backup dates, *etc.*), generated by users and applications,

is typically represented as $\langle attribute, value \rangle$ pairs that describe file properties. Today's storage systems can contain millions to billions of files, and each file can have dozens of metadata attribute-value pairs, resulting in a data set with $10^{10} - 10^{11}$ total pairs.

The ability to search file metadata facilitates complex queries on the properties of files in the storage system, helping administrators understand the kinds of files being stored, where they are located, how they are used, how they got there (provenance), and where they should belong. For example, finding which files to migrate to tape may involve searching file size, access time, and owner metadata attributes, allowing administrators to decide on and enforce their management policies. Metadata search also helps users locate misplaced files, manage their storage space, and track file changes. As a result, metadata search tools are becoming more prevalent; recent reports state that 37% of enterprise businesses use such tools and 40% plan to do so in the near future [12].

To better understand metadata search needs, we surveyed over 30 large scale storage system users and administrators. We found subjects using metadata search for a wide variety of purposes. Use cases included managing storage tiers, tracking legal compliance data, searching large scientific data output files, finding files with incorrect security ACLs, and resource/capacity planning. Table 1 provides examples of some popular use cases and the metadata attributes searched.

2.2 Efficient Metadata Search

Providing efficient metadata search in large-scale storage systems is a challenge. While a number of commercial file metadata search systems exist today [4, 14, 17, 21, 30], these systems focus on smaller scales (*e.g.*, up to tens of millions of files) and are often too slow, resource intensive, and expensive to be effective for large-scale systems. To be effective at large scales, file metadata search must provide the following:

- 1) *Minimal resource requirements.* Metadata search should not require additional hardware. It should be embedded within the storage system and close to the files it indexes while not degrading system performance. Most existing systems require dedicated CPU, memory, and disk hardware, making them expensive and hard to deploy, and limiting their scalability.

- 2) *Fast metadata collection.* Metadata changes must be periodically collected from millions to billions of files without exhausting or slowing the storage system. Existing crawling methods are slow and can tax system resources. Hooks to notify systems of file changes can add overhead to important data paths.

- 3) *Fast and scalable index search and update.* Searches must be fast, even as the system grows, or usability may

File Management Question	Metadata Search Query
Which files can I migrate to tape?	<code>size > 50 GB, atime > 6 months.</code>
How many duplicates of this file are in my home directory?	<code>owner = john, datahash = 0xE431, path = /home/john.</code>
Where are my recently modified presentations?	<code>owner = john, type = (ppt keynote), mtime < 2 days.</code>
Which legal compliance files can be expired?	<code>retention time = expired, mtime > 7 years</code>
Which of my files grew the most in the past week?	<code>Top 100 where size(today) > size(1 week ago), owner = john.</code>
How much storage do these users and applications consume?	<code>Sum size where owner = john, type = database</code>

Table 1: Use case examples. Metadata search use cases collected from our user survey. The high-level questions being addressed are on the left. On the right are the metadata attributes being searched and example values. Users used basic inode metadata, as well as specialized extended attributes, such as legal retention times. Common search characteristics include multiple attributes, localization to part of the namespace, and “back-in-time” search.

suffer. Updates must allow fast periodic re-indexing of metadata. However, existing systems typically rely on general-purpose relational databases (DBMSs) to index metadata. For example, Microsoft’s enterprise search indexes metadata in their Extensible Storage Engine (ESE) database [30]. Unfortunately, DBMSs often use heavy-weight locking and transactions that add overhead even when disabled [43]. Additionally, their designs make significant trade-offs between search and update performance [1]. DBMSs also assume abundant CPU, memory, and disk resources. Although standard DBMSs have benefited from decades of performance research and optimizations, such as vertical partitioning [23] and materialized views, their designs are not a perfect fit for metadata search. This is not a new concept; the DBMS community has argued that general-purpose DBMSs are not a “one size fits all solution” [9, 42, 43], instead saying that application-specific designs are often best.

4) *Easy to use search interface.* Most systems export simple search APIs. However, recent research [3] has shown that specially designed interfaces that can provide an expressive and easy to use query capabilities can greatly improve search experience.

5) *Secure search results.* Search results must not allow users to find or access restricted files [10]. Existing systems either ignore security or enforce it at a significant cost to performance.

We designed Spyglass to address these challenges in large-scale storage systems. Spyglass is specially designed to exploit metadata search properties to achieve scale and performance while being embedded within the storage system. Spyglass focuses on crawling, updating, and searching metadata; interface and security designs are left to future work.

2.3 Metadata Search Properties

To understand metadata search properties, we analyzed results from our user survey and real-world metadata snapshot traces collected from storage servers at NetApp. We then used this analysis to guide our Spyglass design.

Data Set	Description	# of Files	Capacity
Web	web & wiki server	15 million	1.28 TB
Eng	build space	60 million	30 GB
Home	home directories	300 million	76.78 TB

Table 2: Metadata traces collected. The small server capacity of the Eng trace is due to the majority of the files being small source code files: 99% of files are less than 1 KB.

Attribute	Description	Attribute	Description
inumber	inode number	owner	file owner
path	full path name	size	file size
ext	file extension	ctime	change time
type	file or directory	atime	access time
mtime	modification time	hlink	hard link #

Table 3: Attributes used. We analyzed the fields in the inode structure and extracted `ext` values from `path`.

Search Characteristics. From our survey, we observed three important metadata search characteristics. First, over 95% of searches included *multiple metadata attributes* to refine search results; a search on a single attribute over a large file system can return thousands or even millions of results, which users do not want to sift through. Second, about 33% of *searches were localized* to part of the namespace, such as a home or project directory. Users often have some idea of where their files are and a strong idea of where they are not; localizing the search focuses results on only relevant parts of the namespace. Third, about 25% of the searches that users deemed most important *searched multiple versions* of metadata. Users use “back-in-time” searches to understand file trends and how files are accessed.

Metadata Characteristics. We collected metadata snapshot traces from three storage servers at NetApp. Our traces—Web, Eng, and Home—are described in Table 2. Table 3 describes the metadata attributes that we analyzed. NetApp servers support extended attributes, though they were rarely used in these traces. We found two key properties in these traces: metadata has *spatial locality* and highly *skewed distributions* of values.

Spatial locality means that attribute values are clustered in the namespace (*i.e.*, occurring in relatively few directories). For example, `john`’s files reside mostly in

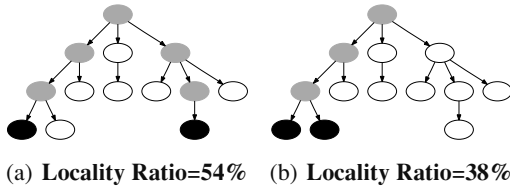


Figure 1: Examples of locality ratio. Directories that recursively contain the `ext` attribute value `html` are black and gray. The black directories contain the value. The locality ratio of `ext` value `html` is 54% ($= 7/13$) in the first tree and 38% ($= 5/13$) in the second tree. The value of `html` has better spatial locality in the second tree than in the first one.

the `/home/john` sub-tree, not scattered evenly across the namespace. Spatial locality comes from the way that users and applications organize files in the namespace, and has been noted in other file system studies [2, 25]. To measure spatial locality, we use an attribute value’s *locality ratio*: the percent of directories that recursively contain the value, as illustrated in Figure 1. A directory recursively contains an attribute value if it or any of its sub-directories contains the value. The figure on the right has a lower locality ratio because the `ext` attribute value `html` is recursively contained in fewer directories. Table 4 shows the locality ratios for the 32 most frequently occurring values for various attributes (`ext`, `size`, `owner`, `ctime`, `mtime`) in each trace. Locality ratios are less than 1% for all attributes, meaning that 99% of directories do not recursively contain the value. We expect extended attributes to exhibit similar properties since they are often tied to file type and owner attributes.

Utilizing spatial locality can help prune a query’s search space by identifying only the parts of the namespace that contain a metadata value, eliminating a large number of files to search. Unfortunately, most general-purpose DBMSs treat path names as flat string attributes, making it difficult for them to utilize this information, instead typically requiring them to consider *all* files for a search no matter its locality.

Metadata values also have highly skewed frequencies—their popularity distributions are asymmetric, causing a few very popular metadata values to account for a large fraction of all total values. This distribution has also been observed in other metadata studies [2, 11]. Figures 2(a) and 2(b) show the distribution of `ext` and `size` values from our Home trace on a log-log scale. The linear appearance indicates that the distributions are Zipf-like and follow the power law distribution [40]. In these distributions, 80% of files have one of the 20 most popular `ext` or `size` values, while the remaining 20% of the files have thousands of other values. Figure 2(c) shows the distribution of the Cartesian product (*i.e.*, the intersection) of the top 20 `ext` and `size` values. The curve is much flatter, which indicates a more even distribution of values. Only 33%

of files have one of the top 20 `ext` and `size` combinations. In Figure 2(c), file percentages for corresponding ranks are at least an order of magnitude lower than in the other two graphs. This means, for example, that there are many files with `owner john` and many files with `ext pdf`, but there are often over an order of magnitude fewer files with *both* `owner john` and `ext pdf`.

These distribution properties show that multi-attribute searches will significantly reduce the number of query results. Unfortunately, most DBMSs rely on attribute value distributions (also known as selectivity) to choose a query plan. When distributions are skewed, query plans often require extra data processing [28]; for example, they may retrieve all of `john`’s files to find the few that are `john`’s `pdf` files or vice-versa. Our analysis shows that query execution should utilize attribute values’ spatial locality rather than their frequency distributions. Spatial locality provides a more effective way to execute a query because it is more selective and can better reduce a query’s search space.

3 Spyglass Design

Spyglass uses several novel techniques that exploit the metadata search properties discussed in Section 2 to provide fast, scalable search in large-scale storage systems. First, *hierarchical partitioning* partitions the index based on the namespace, preserving spatial locality in the index and allowing fine-grained index control. Second, *signature files* [13] are used improve search performance by leveraging locality to identify only the partitions that are relevant to a query. Third, *partition versioning* versions index updates, which improves update performance and allows “back-in-time” search of past metadata versions. Finally, Spyglass utilizes storage systems snapshots to crawl only the files whose metadata has changed, providing fast collection of metadata changes. Spyglass resides within the storage system and consists of two major components, shown in Figure 3: the Spyglass index, which stores metadata and serves queries, and a crawler that extracts metadata from the storage system.

3.1 Hierarchical Partitioning

To exploit metadata locality and improve scalability, the Spyglass index is partitioned into a collection of separate, smaller indexes, which we call hierarchical partitioning. Hierarchical partitioning is based on the storage system’s namespace and encapsulates separate parts of the namespace into separate partitions, thus allowing more flexible, finer grained control of the index. Similar partitioning strategies are often used by file systems to distribute the namespace across multiple machines [35, 44].

	ext	size	uid	ctime	mtime
Web	0.000162% – 0.120%	0.0579% – 0.177%	0.000194% – 0.0558%	0.000291% – 0.0105%	0.000388% – 0.00720%
Eng	0.00101% – 0.264%	0.00194% – 0.462%	0.000578% – 0.137%	0.000453% – 0.0103%	0.000528% – 0.0578%
Home	0.000201% – 0.491%	0.0259% – 0.923%	0.000417% – 0.623%	0.000370% – 0.128%	0.000911% – 0.0103%

Table 4: Locality ratios of the 32 most frequently occurring attribute values. All locality ratios are well below 1%, which means that files with these attribute values are recursively contained in less than 1% of directories.

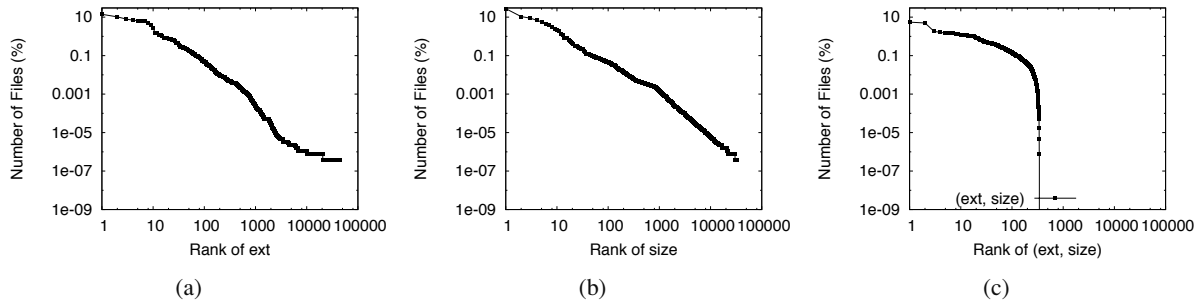


Figure 2: Attribute value distribution examples. A rank of 1 represents the attribute value with the highest file count. The linear curves on the log-log scales in Figures 2(a) and 2(b) indicate a Zipf-like distribution, while the flatter curve in Figure 2(c) indicates a more even distribution.

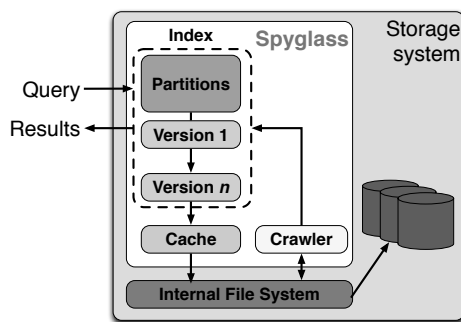


Figure 3: Spyglass overview. Spyglass resides within the storage system. The crawler extracts file metadata, which gets stored in the index. The index consists of a number of partitions and versions, all of which are managed by a caching system.

Each of the Spyglass partitions is stored sequentially on disk, as shown in Figure 4. Thus, unlike a DBMS, which stores records adjacently on disk using their row or column order, Spyglass groups records nearby in the namespace together on disk. This approach improves performance since the files that satisfy a query are often clustered in only a portion of the namespace, as shown by our observations in Section 2. For example, a search of the storage system for john’s .ppt files likely does not require searching sub-trees such as other user’s home directories or system file directories. Hierarchical partitioning allows only the sub-trees relevant to a search to be considered, thereby enabling reduction of the search space and improving scalability. Also, a user may choose to localize the search to only a portion of the namespace. Hierarchical partitioning allows users to control the scope of the files that are searched. A DBMS-based

solution usually encodes pathnames as flat strings, making it oblivious to the hierarchical nature of file organization and requiring it to consider the entire namespace for each search. If the DBMS stores the files sorted by file name, it can improve locality and reduce the fraction of the index table that must be scanned; however, this approach can still result in performance problems for index updates, and does not encapsulate the hierarchical relationship between files.

Spyglass partitions are kept small, on the order of 100,000 files, to maintain locality in the partition and to ensure that each can be read and searched very quickly. Since partitions are stored sequentially on disk, searches can usually be satisfied with only a few small sequential disk reads. Also, sub-trees often grow at a slower rate than the system as a whole [2, 25], which provides scalability because the number of partitions to search will often grow slower than the size of the system.

We refer to each partition as a *sub-tree partition*; the Spyglass index is a tree of sub-tree partitions that reflects the hierarchical ordering of the storage namespace. Each partition has a main *partition index*, in which file metadata for the partition is stored; *partition metadata*, which keeps information about the partition; and pointers to child partitions. Partition metadata includes information used to determine if a partition is relevant to a search and information used to support partition versioning.

The Spyglass index is stored persistently on disk; however, all partition metadata, which is small, is cached in-memory. A *partition cache* manages the movement of entire partition indexes to and from disk as needed. When a file is accessed, its neighbor files will likely need to be accessed as well, due to spatial locality. Paging en-

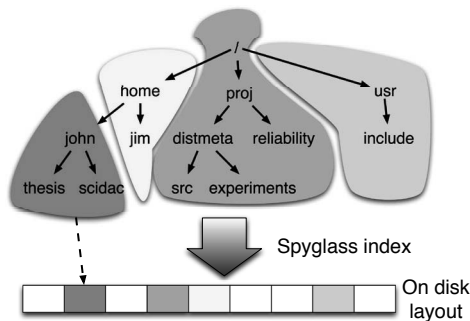


Figure 4: Hierarchical partitioning example. Sub-tree partitions, shown in different colors, index different storage system sub-trees. Each partition is stored sequentially on disk. The Spyglass index is a tree of sub-tree partitions.

partition indexes allows metadata for all of these files to be fetched in a single, small sequential read. This concept is similar to the use of embedded inodes [15], to store inodes adjacent to their parent directory on disk.

In general, Spyglass search performance is a function of the number of partitions that must be read from disk. Thus, the partition cache's goal is to reduce disk accesses by ensuring that most partitions searched are already in-memory. While we know of no studies of file system query patterns we believe that a simple LRU algorithm is effective. Both web queries [5] and file system access patterns [25] exhibit skewed, Zipf-like popularity distributions, suggesting that file metadata queries *may* exhibit similar popularity distributions; this would mean that only a small subset of partitions will be frequently accessed. An LRU algorithm keeps frequently accessed partitions in-memory, ensuring high performance for common queries and efficient cache utilization.

Partition Indexes. Each partition index must provide fast, multi-dimensional search of the metadata it indexes. To do this we use a K-D tree [7], which is a k -dimensional binary tree, because it provides lightweight, logarithmic point, range, and nearest neighbor search over k dimensions and allows multi-dimensional search of a partition in tens to hundreds of microseconds. However, other index structures can provide additional functionality. For example, FastBit [45] provides high index compression, Berkeley DB [34] provides transactional storage, cache-oblivious B-trees [6] improve update time, and K-D-B-trees [38] allow partially in-memory K-D trees. However, in most cases, the fast, lightweight nature of K-D trees is preferred. The drawback is that K-D trees are difficult to update; Section 3.2 describes techniques to avoid continuous updates.

Partition Metadata. Partition metadata contains information about the files in the partition, including paths of indexed sub-trees, file statistics, signature files, and version information. File statistics, such as file counts and minimum and maximum values, are kept to answer

aggregation and trend queries without having to process the entire partition index. These statistics are computed as files are being indexed. A *version vector*, which is described in Section 3.2, manages partition versions. Signature files are used to determine if the partition contains files relevant to a query.

Signature files [13] are bit arrays that serve as compact summaries of a partition's contents and exploit metadata locality to prune a query's search space. A common example of a signature file is the Bloom Filter [8]. Spyglass can determine whether a partition *may* index any files that match a query simply by testing bits in the signature files. A signature file and an associated hashing function are created for each attribute indexed in the partition. All bits in the signature file are initially set to zero. As files are indexed, their attribute values are hashed to a bit position in the attribute's signature file, which is set to one. To determine if the partition indexes files relevant to a query, each attribute value being searched is hashed and its bit position is tested. The partition needs to be searched *only* if *all* bits tested are set to one. Due to spatial locality, most searches can eliminate many partitions, reducing the number of disk accesses and processing a query must perform.

Due to collisions in the hashing function that cause false positives, a signature file determines only if a partition *may* contain files relevant to a query, potentially causing a partition to be searched when it does not contain any files relevant to a search. However, signature files cannot produce false negatives, so partitions with relevant files will never be missed. False-positive rates can be reduced by varying the size of the signature or changing the hashing function. Increasing signature file sizes, which are initially around 2 KB, decreases the chances of a collision by increasing the total number of bits. This trades off increased memory requirements and lower false positive rates. Changing the hashing function allow a bit's meaning and how it is used to be improved. For example, consider a signature file for file size attributes. Rather than have each bit represent a single size value (*e.g.*, 522 bytes), we can reduce false positives for common small files by mapping each 1 KB range to a single bit for sizes under 1 MB. The ranges for less common large files can be made more coarse, perhaps using a single bit for sizes between 25 and 50 MB.

The number of signature files that have to be tested can be reduced by utilizing the tree structure of the Spyglass index to create hierarchically defined signature files. Hierarchical signature files are smaller signatures (roughly 100 bytes) that summarize the contents of its partition and the partitions below it in the tree. Hierarchical signature files are the logical OR of a partition's signature files and the signature files of its children. A single failed test of a hierarchical signature file can eliminate huge parts of

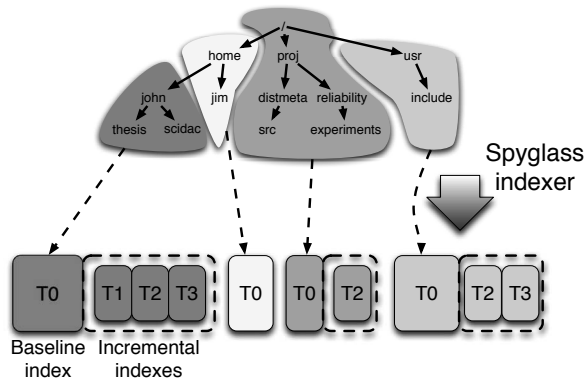


Figure 5: Versioning partitioning example. Each sub-tree partition manages its own versions. A baseline index is a normal partition index from some initial time T_0 . Each incremental index contains the changes required to roll query result forward to a new point in time. Each sub-tree partition manages its version in a version vector.

the index from the search space, preventing every partition’s signature files from being tested. Hierarchical signature files are kept small to save memory at the cost of increased false positives.

3.2 Partition Versioning

Spyglass improves update performance and enables “back-in-time” search using a technique called partition versioning that batches index updates, treating each batch as a new incremental index version. The motivation for partition versioning is two-fold. First, we wish to improve index update performance by not having to modify existing index structures. In-place modification of existing indexes can generate large numbers of disk seeks and can cause partition index structures to become unbalanced. Second, back-in-time search can help answer many important storage management questions that can track file trends and how they change.

Spyglass batches updates before they are applied as new versions to the index, meaning that the index may be stale because file modifications are not immediately reflected in the index. However, batching updates improves index update performance by eliminating many small, random, and frequent updates that can thrash the index and cache. Additionally, from our user survey, most queries can be satisfied with a slightly stale index. It should be noted that partition versioning does not require updates to be batched. The index can be updated in real time by versioning each individual file modification, as is done in most versioning file systems [39, 41].

Creating Versions. Spyglass versions each sub-tree partition individually rather than the entire index as a whole in order to maintain locality. A versioned sub-tree partition consists of two components: a *baseline index* and

incremental indexes, which are illustrated in Figure 5. A baseline index is a normal partition index that represents the state of the storage system at time T_0 , or the time of the initial update. An incremental index is an index of metadata *changes* between two points in time T_{n-1} and T_n . These changes are indexed in K-D trees, and smaller signature files are created for each incremental index. Storing changes differs from the approach used in some versioning file systems [39], which maintain full copies for each version. Changes consist of metadata creations, deletions, and modifications. Maintaining only changes requires a minimal amount of storage overhead, resulting in a smaller footprint and less data to read from disk.

Each sub-tree partition starts with a baseline index, as shown in Figure 5. When a batch of metadata changes is received at T_1 , it is used to build incremental indexes. Each partition manages its incremental indexes using a *version vector*, similar in concept to inode logs in the Elephant File System [39]. Since file metadata in different parts of the file system change at different rates [2, 25], partitions may have different numbers and sizes of incremental indexes. Incremental indexes are stored sequentially on disk adjacent to their baseline index. As a result, updates are fast because each partition writes its changes in a single, sequential disk access. Incremental indexes are paged into memory whenever the baseline index is accessed, increasing the amount of data that must be read when paging in a partition, though not typically increasing the number of disk seeks. As a result, the overhead of versioning on overall search performance is usually small.

Performing a “back-in-time” search that is accurate as of time T_n works as follows. First, the baseline index is searched, producing query results that are accurate as of T_0 . The incremental indexes T_1 through T_n are then searched in chronological order. Each incremental index searched produces metadata changes that modify the search results, rolling them forward in time, and eventually generating results that are accurate as of T_n . For example, consider a query for files with **owner john** that matches two files, F_a and F_b , at T_0 . A search of incremental indexes at T_1 may yield changes that cause F_b to no longer match the query (e.g., no longer owned by john), and a later search of incremental indexes at T_n may yield changes that cause file F_c to match the query (i.e., now owned by john). The results of the query are F_a and F_c , which is accurate as of T_n . Because this process is done in memory and each version is relatively small, searching through incremental indexes is often very fast. In rolling results forward, a small penalty is paid to search the most recent changes; however, updates are much faster because no data needs to be copied, as is the case in CVFS [41], which rolls version changes backwards rather than forwards.

Managing Versions. Over time, older versions tends to decrease in value and should be removed to reduce search overhead and save space. Spyglass provides two efficient techniques for managing partition versions: *version collapsing* and *version checkpointing*. Version collapsing applies each partition’s incremental index changes to its baseline index. The result is a single baseline for each partition that is accurate as of the most recent incremental index. Collapsing is efficient because all original index data is read sequentially and the new baseline is written sequentially. Version checkpointing allows an index to be saved to disk as a new copy to preserve an important landmark version of the index.

We describe how collapsing and checkpointing can be used with an example. During the day, Spyglass is updated hourly, creating new versions every hour, thus allowing “back-in-time” searches to be performed at per-hour granularity over the day. At the end of each day, incremental versions are collapsed, reducing space overhead at the cost of prohibiting hour-by-hour searching over the last day. Also, at the end of each day, a copy of the collapsed index is checkpointed to disk, representing the storage system state at the end of each day. At the end of each week, all but the latest daily checkpoints are deleted; and at the end of each month, all but the latest weekly checkpoints are deleted. This results in versions of varying time scales. For example, over the past day any hour can be searched, over the past week any day can be searched, and over the past month any week can be searched. The frequency for index collapsing and checkpointing can be configured based on user needs and space constraints.

3.3 Collecting Metadata Changes

The Spyglass crawler takes advantage of NetApp Snapshot™ technology in the NetApp WAFL® file system [19] on which it was developed to quickly collect metadata changes. Given two snapshots, T_{n-1} and T_n , Spyglass calculates the difference between them. This difference represents all of the file metadata changes between T_{n-1} and T_n . Because of the way snapshots are created, only the metadata of *changed* files is re-crawled.

All metadata in WAFL resides in a single file called the *inode file*, which is a collection of fixed length inodes. Extended attributes are included in the inodes. Performing an initial crawl of the storage system is fast because it simply involves sequentially reading the inode file. Snapshots are created by making a copy-on-write clone of the inode file. Calculating the difference between two snapshots leverages this mechanism. This is shown in Figure 6. By looking at the block numbers of the inode file’s indirect and data blocks, we can determine exactly which blocks have changed. If a block’s number has not

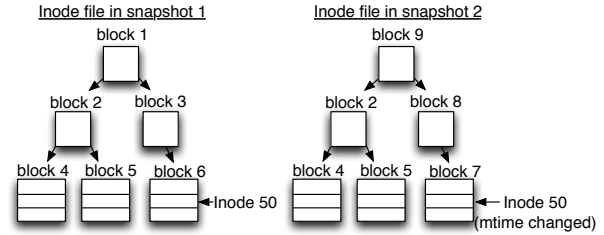


Figure 6: Snapshot-based metadata collection. In snapshot 2, block 7 has changed since snapshot 1. This change is propagated up the tree. Because block 2 has not changed, we do not need to examine it or any blocks below it.

changed, then it does not need to be crawled. If this block is an indirect block, then no blocks that it points to need to be crawled either because block changes will propagate all the way back up to the inode file’s root block. As a result, the Spyglass crawler can identify just the data blocks that have changed and crawl only their data. This approach greatly enhances scalability because crawl performance is a function of the number of files that have changed rather than the total number of files.

Spyglass is not dependent on snapshot-based crawling, though it provides benefits compared to alternative approaches. Periodically walking the file system can be extremely slow because each file must be traversed. Moreover, traversal can utilize significant system resources and alter file access times on which file caches depend. Another approach, file system event notifications (*e. g.*, *inotify* [22]), requires hooks into critical code paths, potentially impacting performance. A changelog, such as the one used in NTFS, is another alternative; however, since we are not interested in every system event, a snapshot-based scheme is more efficient.

3.4 Distributed Design

Our discussion thus far has focused on indexing and crawling on a single storage server. However, large-scale storage systems are often composed of tens or hundreds of servers. While we do not currently address how to distribute the index, we believe that hierarchical partitioning lends itself well to a distributed environment because the Spyglass index is a tree of partitions. A distributed file system with a single namespace can view Spyglass as a larger tree composed of partitions placed on multiple servers. As a result, distributing the index is a matter of effectively scaling the Spyglass index tree. Also, the use of signature files may be effective at routing distributed queries to relevant servers and their sub-trees. Obviously, there are many challenges to actually implementing this. A complete design is left to future work.

4 Experimental Evaluation

We evaluated our Spyglass prototype to determine how well our design addresses the metadata search challenges described in Section 2 for varying storage system sizes. To do this, we first measured metadata collection speed, index update performance, and disk space usage. We then analyzed search performance and how effectively index locality is utilized. Finally, we measured partition versioning overhead.

Implementation Details. Our Spyglass prototype was implemented as a user-space process on Linux. An RPC-based interface to WAFL gathers metadata changes using our snapshot-based crawler. Our prototype dynamically partitions the index as it is being updated. As files and directories are inserted into the index, they are placed into the partition with the longest pathname match (*i.e.*, the pathname match farthest down the tree). New partitions are created when a directory is inserted and all matching partitions are full. A partition is considered full when it contains over 100,000 files. We use 100,000 as the soft partition limit in order to ensure that partitions are small enough to be efficiently read and written to disk. Using a much smaller partition size will often increase the number of partitions that must be accessed for a query; this incurs extra expensive disk seeks. Using a much larger partition size decreases the number of partitions that must be accessed for a query; however it poorly encapsulates spatial locality, causing extra data to be read from disk. In the case of symbolic and hard links, multiple index entries are used for the file.

During the update process, partitions are buffered in-memory and written sequentially to disk when full; each is stored in a separate file. K-D trees were implemented using `libkdtree++` [27]. Signature file bit-arrays are about 2 KB, but *hierarchical* signature files are only 100 bytes, ensuring that signature files can fit within our memory constraints. Hashing functions that allowed each signature file's bit to correspond to a range of values were used for file size and time attributes to reduce false positive rates. When incremental indexes are created, they are appended to their partition on disk. Finally, we implement a simple search API that allows point, range, top-*k*, and aggregation searches. We plan to extend this interface as future work.

Experimental Setup. We evaluated performance using our real-world metadata traces described in Table 2. These traces have varying sizes, allowing us to examine scalability. Our Web and Eng traces also have incremental snapshot traces of daily metadata changes for several days. Since no standard benchmarks exist, we constructed synthetic sets of queries, discussed later in this section, from our metadata traces to evaluate search performance. All experiments were performed on a dual

core AMD Opteron machine with 8 GB of main memory running Ubuntu Linux 7.10. All index files were stored on a network partition that accessed a high-end NetApp file server over NFS.

We also evaluated the performance of two popular relational DBMSs, PostgreSQL and MySQL, which serve as relative comparison points to DBMS-based solutions used in other metadata search systems. The goal of our comparison is to provide some context to frame our Spyglass evaluation, not to compare performance to the best possible DBMS setup. We compared Spyglass to an index-only DBMS setup, which is used in several commercial metadata search systems, and also tuned various options, such as page size, to the best of our ability. This setup is effective at pointing out several basic DBMS performance problems. DBMS performance *can* be improved through the techniques discussed in Section 2; however, as stated earlier, they do not completely match metadata search cost and performance requirements.

Our Spyglass prototype indexes the metadata attributes listed in Table 3. Our index-only DBMSs include a base relation with the same metadata attributes and a B+-tree index for each. Each B+-tree indexes table row ID. An index-only design reduces space usage compared to some more advanced setups, though it has slower search performance. In all three traces, cache sizes were configured to 128 MB, 512 MB, and 2.5 GB for the Web, Eng, and Home traces, respectively. These sizes are small relative to the size of their trace and correspond to about 1 MB for every 125,000 files.

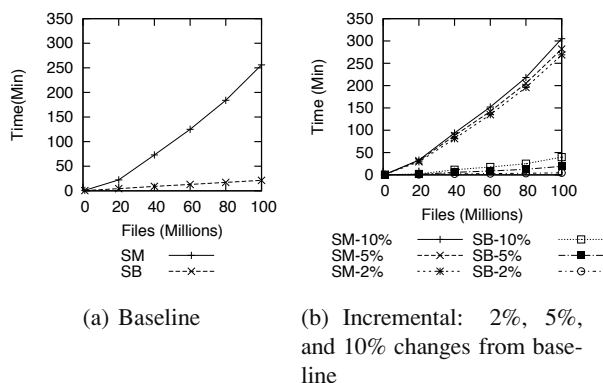


Figure 7: Metadata collection performance. We compare Spyglass's snapshot-based crawler (SB) to a straw-man design (SM). Our crawler has good scalability; performance is a function of the number of changed files rather than system size.

Metadata Collection Performance. We first evaluated our snapshot-based metadata crawler and compared it to a straw-man approach. Fast collection performance impacts how often updates occur and system resource utilization. Our straw-man approach performs a parallelized walk of the file system using `stat()` to ex-

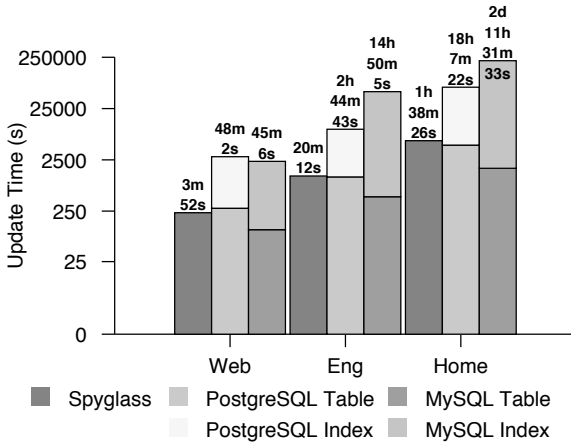


Figure 8: Update performance. The time required to build an initial baseline index shown on a log-scale. Spyglass updates quickly and scales linearly because updates are written to disk mostly sequentially.

tract metadata. Figure 7(a) shows the performance of a baseline crawl of all file metadata. Our snapshot based crawler is up to $10\times$ faster than our straw-man for 100 million files because our approach simply scans the inode file. As a result, a 100 million file system is crawled in less than 20 minutes.

Figure 7(b) shows the time required to collect incremental metadata changes. We examine systems with 2%, 5%, and 10% of their files changed. For example, a baseline of 40 million files and 5% change has 2 million changed files. For the 100 million file tests, each of our crawls finishes in under 45 minutes, while our straw-man takes up to 5 hours. Our crawler is able to crawl the inode file at about 70,000 files per second. Our crawler effectively scales because we incur only a fractional overhead as more files are crawled; this is due to our crawling only changed blocks of the inode file.

Update Performance. Figure 8 shows the time required to build the initial index for each of our metadata traces. Spyglass requires about 4 minutes, 20 minutes, and 100 minutes for the three traces, respectively. These times correspond to a rate of about 65,000 files per second, indicating that update performance scales linearly. Linear scaling occurs because updates to each partition are written sequentially, with seeks occurring only between partitions. Incremental index updates have a similar performance profile because metadata changes are written in the same fashion and few disk seeks are added. Our reference DBMSs take between $8\times$ and $44\times$ longer to update because DBMSs require loading their base table and updating index structures. While loading the table is fast, updating index structures often requires seeks back to the base table or extra data copies. As a result, DBMS updates with our Home trace can take a day or

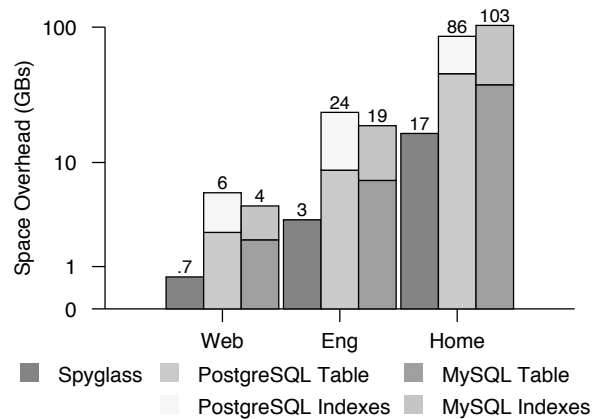


Figure 9: Space overhead. The index disk space requirements shown on a log-scale. Spyglass requires just 0.1% of the Web and Home traces and 10% of the Eng trace to store the index.

more; however, approaches such as cache-oblivious B-trees [6] may be able to reduce this gap.

Space Overhead. Figure 9 shows the disk space usage for all three of our traces. Efficient space usage has two primary benefits: less disk space taken from the storage system and the ability to cache a higher fraction of the index. Spyglass requires less than 0.1% of the total disk space for the Web and Home traces. However, it requires about 10% for the Eng trace because the total system size is low due to very small files. Spyglass requires about 50 bytes per file across all traces, resulting in space usage that scales linearly with system size. Space usage in Spyglass is $5\times$ – $8\times$ lower than in our references DBMSs because they require space to store the base table and index structures. Figure 9 shows that building index structures can more the double the total space requirements.

Search Performance. To evaluate Spyglass search performance, we generated sets of queries derived from real-world queries in our user study; there are, unfortunately, no standard benchmarks for file system search. These query sets are summarized in Table 5. Our first set is based on a storage administrator searching for the user and application files that are consuming the most space (e.g., total size of john’s .vmdk files)—an example of a simple two-attribute search. The second set is an administrator localizing the same search to only part of the namespace, which shows how localizing the search changes performance. The third set is a storage user searching for recently modified files of a particular type in a specific sub-tree, demonstrating how searching many attributes impacts performance. Each query set consists of 100 queries, with attribute values randomly selected from our traces. Randomly selecting attribute values means that our query sets loosely follow the distribution of values in our traces and that a variety of values are used.

Set	Search	Metadata Attributes
Set 1	Which user and application files consume the most space?	Sum sizes for files using <code>owner</code> and <code>ext</code> .
Set 2	How much space, in this part of the system, do files from query 1 consume?	Use query 1 with an additional directory <code>path</code> .
Set 3	What are the recently modified application files in my home directory?	Retrieve all files using <code>mtime</code> , <code>owner</code> , <code>ext</code> , and <code>path</code> .

Table 5: Query Sets. A summary of the searches used to generate our evaluation query sets.

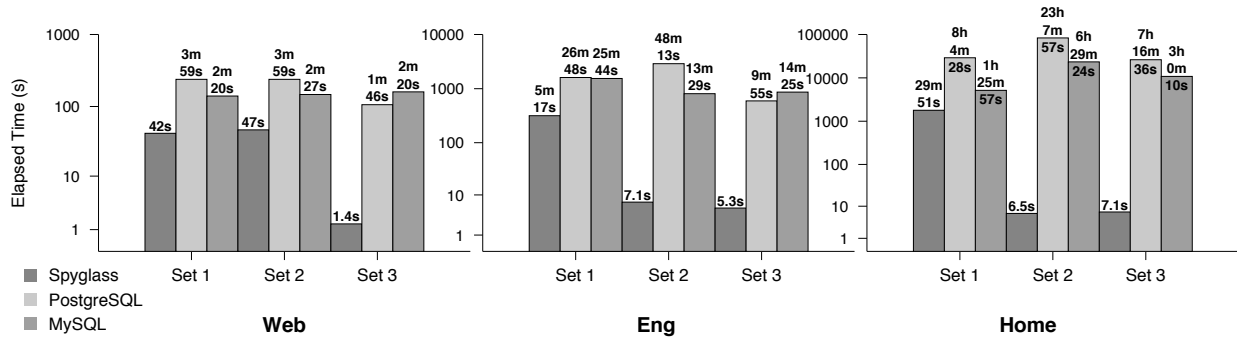


Figure 10: Query set run times. The total time required to run each set of queries. Each set is labeled 1 through 3 and is clustered by trace file. Each trace is shown on a separate log-scale axis. Spyglass improves performance by reducing the search space to a small number of partitions, especially for query sets 2 and 3, which are localized to only a part of the namespace.

Figure 10 shows the total run times for each set of queries. In general, query set 1 takes Spyglass the longest to complete, while query sets 2 and 3 finish much faster. This performance difference is caused by the ability of sets 2 and 3 to localize the search to only a part of the namespace by including a path with the query. Spyglass is able to search only files from this part of the storage system by using hierarchical partitioning. As a result, the search space for these queries is bound to the size of the sub-tree, no matter how large the storage system. Because the search space is already small, using many attributes has little impact on performance for set 3. Query set 1, on the other hand, must consider all partitions and tests each partition’s signature files to determine which to search. While many partitions are eliminated, there are more partitions to search than in the other query sets, which accounts for the longer run times.

Our comparison DBMSs perform closer to Spyglass on our smallest trace, Web; however we see the gap widen as the system size increases. In fact, Spyglass is over four orders of magnitude faster for query sets 2 and 3 on our Home trace, which is our largest at 300 million files. The large performance gap is due to several reasons. First, our DBMSs consider files from all parts of the namespace, making the search space much larger. Second, skewed attribute value distributions cause our DBMSs to process extra data even when there are few results. Third, the DBMSs base tables ignore metadata locality, causing extra disk seeks to find files close in the namespace but far apart in the table. Spyglass, on the other hand, uses hierarchical partitioning to significantly reduce the search space, performs only small, sequential

disk accesses, and can exploit locality in the workload to greatly improve cache utilization.

Using the results from Figure 10, we calculated query throughput, shown in Table 6. Query throughput (queries per second) provides a normalized view of our results and the query loads that can be achieved. Spyglass achieves throughput of multiple queries per second in all but two cases; in contrast, the reference DBMSs do not achieve one query per second in any instance, and, in many cases, cannot even sustain one query per five minutes. Figure 11 shows an alternate view of performance; a cumulative distribution function (CDF) of query execution times on our Home trace, allowing us to see how each query performed. In query sets 2 and 3, Spyglass finishes all searches in less than a second because localized searches bound the search space. For query set 1, we see that 75% of queries take less than one second, indicating that most queries are fast and that a few slow queries contribute significantly to the total run times in Figure 10. These queries take longer because they must read many partitions from disk, either because few were previously cached or many partitions are searched.

Index Locality. We now evaluate how well Spyglass exploits spatial locality to improve query performance. We generated another set of queries, based on query 1 from Table 5, with 500 queries with `owner` and `ext` values randomly selected from our Eng trace. We generated similar query sets for individual `ext` and `owner` attributes.

Figure 12(a) shows a CDF of the fraction of partitions searched. Searching more partitions often increases the amount of data that must be read from disk, which decreases performance. We see that 50% of searches using just the `ext` attribute reference fewer than 75% of par-

System	Web			Eng			Home		
	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3
Spyglass	2.38	2.12	71.4	0.315	14.1	18.9	0.05	15.4	14.1
PostgreSQL	0.418	0.418	0.94	0.062	0.034	0.168	0.003	0.001	0.003
MySQL	0.714	0.68	0.063	0.647	0.123	0.115	0.019	0.004	0.009

Table 6: Query throughput. We use the results from Figure 10 to calculate query throughput (queries per second). We find that Spyglass can achieve query throughput that enables fast metadata search even on large-scale storage systems.

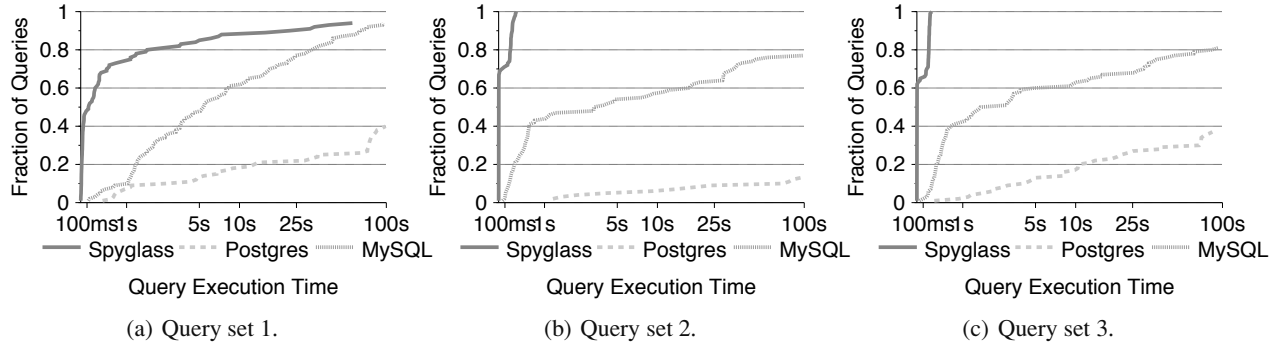


Figure 11: Query execution times. A CDF of query set execution times for the Eng trace. In Figures 11(b) and 11(c), all queries are extremely fast because these sets include a path predicate that allows Spyglass to narrow the search to a few partitions.

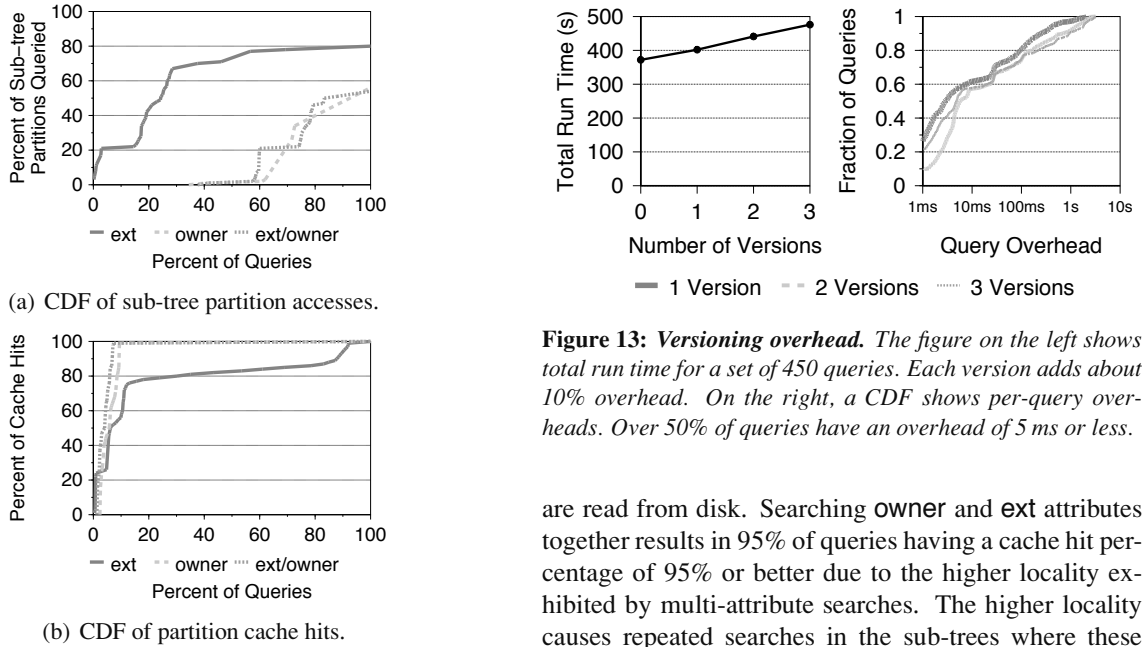


Figure 12: Index locality. A CDF of the number of partitions accessed and the number of accesses that were cache hits for our query set. Searching multiple attributes reduces the number of partition accesses and increases cache hits.

tions. However, 50% of searches using both `ext` and `owner` together reference fewer than 2% of the partitions, since searching more attributes increases the locality of the search, thereby reducing the number of partitions that must be searched. Figure 12(b) shows a CDF of cache hit percentages for the same set of queries. Higher cache hit percentages means that fewer partitions

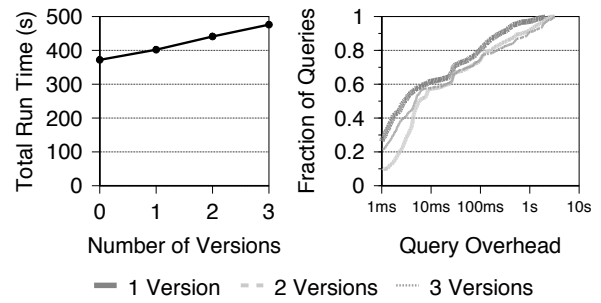


Figure 13: Versioning overhead. The figure on the left shows total run time for a set of 450 queries. Each version adds about 10% overhead. On the right, a CDF shows per-query overheads. Over 50% of queries have an overhead of 5 ms or less.

are read from disk. Searching `owner` and `ext` attributes together results in 95% of queries having a cache hit percentage of 95% or better due to the higher locality exhibited by multi-attribute searches. The higher locality causes repeated searches in the sub-trees where these files reside and allows Spyglass to ignore more non-relevant partitions.

Versioning Overhead. To measure the search overhead added by partition versioning, we generated 450 queries based on query 1 from Table 5 with values randomly selected from our Web trace. We included three full days of incremental metadata changes, and used them to perform three incremental index updates. Figure 13 shows the time required to run our query set with an increasing number of versions; each version adds about a 10% overhead to the total run time. However, the overhead added to most queries is quite small. Figure 13 also shows, via

a CDF of the query overheads incurred for each version, that more than 50% of the queries have less than a 5 ms overhead. Thus, it is a few much slower queries that contribute to most of the 10% overhead. This behavior occurs because overhead is typically incurred when incremental indexes are read from disk, which doesn't occur once a partition is cached. Since reading extra versions does not typically incur extra disk seeks, the overhead for the slower queries is mostly due to reading partitions with much larger incremental indexes from disk.

5 Related Work

Spyglass seeks to improve how file systems manage growing volumes of data, which has been an important challenge and an active area of research for over two decades. A significant amount of work has looked at how file systems can improve file naming and organization by leveraging file attributes. The Semantic File System [16] utilized file $\langle \text{attribute}, \text{value} \rangle$ pairs to dynamically construct a namespace based on queries rather than use a standard hierarchical namespace. Virtual directories allowed queries to be integrated directly into the namespace as a directory containing search results. The Hierarchy and Content (HAC) [18] file system looked at how Semantic File System concepts could be applied to a hierarchical namespace, providing users with a new naming mechanism without requiring them to forgo traditional hierarchies. These and similar systems [32, 36] focus on how users name and view files, though they do not focus on how files are actually indexed and searched, thereby potentially limiting their performance and scalability. While Spyglass does not provide higher level naming semantics, it is the first to address the challenge of scalable file metadata indexing and search, allowing it to potentially be used as the underlying indexing method for such file systems.

Spyglass focuses on how to exploit file metadata properties to improve search performance and scalability, though it is not the first to look at how new indexing structures improve file retrieval. Inversion [33] used a general-purpose DBMS as the core file system structure, rather than traditional file system inode and data layouts. Inversion used several PostgreSQL tables to store both file metadata and data, allowing the file system to benefit from database transaction and recovery support and allowing metadata and data to be queried. Like Spyglass, Inversion provides ad hoc metadata query functionality, though it focuses on allowing file systems to leverage database functionality rather than on query performance.

However, a number of new index designs have been proposed to improve various aspects of file system search. GLIMPSE [29] reduced disk space requirements, compared to a normal full-text inverted index, by main-

taining only a partial inverted index that does not store the location of every term occurrence. Like Spyglass, GLIMPSE partitioned the search space, using fixed size blocks of the file space, which were then referenced by the partial inverted index. A tool similar to `grep` was used to find exact term locations with each fixed size block. Similarly, Diamond [20] eliminated disk space requirements by using a mechanism to improve the speed of brute force searches instead of maintaining an index. A technique called Early Discard allowed files that are irrelevant to the search to be rejected as early as possible, helping to reduce the search space. Early Discard used application-specific “searchlets” to determine when a file is irrelevant to a given query. Geometric partitioning [24] aimed to improve inverted index update performance by breaking up the inverted index's inverted lists by update time. The most recently updated inverted lists were kept small and sequential, allowing future updates to be applied quickly. A merging algorithm created new partitions as the lists grow over time. Query-based partitioning [31] used a similar approach, though it partitioned the inverted index based on file search frequency, allowing index data for infrequently searched files to be offloaded to second-tier storage to improve cost.

6 Conclusions and Future Work

As storage systems have become larger, finding and managing files has become increasingly difficult. To address this problem we presented Spyglass, a metadata search system that improves file management by allowing complex, ad hoc queries over file metadata. Spyglass introduces several novel indexing techniques that improve metadata crawling, search, and update performance by exploiting metadata properties. Our evaluation shows that Spyglass has up to 1–4 orders of magnitude faster search performance than existing designs.

We plan on improving Spyglass in the future in a number of ways. First, we plan on addressing file security by leveraging hierarchical partitioning to help eliminate partitions that the user does not have access to from the search space. Second, we are exploring new interface and query language designs that allow users to ask complex queries (*e.g.*, “back-in-time” queries) while remaining easy to use. Third, we propose fully distributing Spyglass across a cluster by allowing partitions to be replicated and migrated across machines. Fourth, we will explore how partitioning can be improved by using other metadata attributes to partition the index. Finally, we are looking at how Spyglass can be used as the main metadata store for a storage system, eliminating many of the space and performance overheads incurred when used in addition to the storage system's metadata store.

Acknowledgments

We would like to thank our colleagues in the Storage Systems Research Center and NetApp's Advanced Technology Group for their input and guidance. Also, we thank Remzi Arpaci-Dusseau, Stavros Harizopoulos, and Jiri Schindler for their early feedback and discussions on this work. Finally, we thank our shepherd Sameer Ajmani and our anonymous reviewers, whose comments significantly improved the quality of this paper.

This work was supported in part by the Department of Energy's Petascale Data Storage Institute under award DE-FC02-06ER25768 and by the National Science Foundation under award CCF-0621463. We thank the industrial affiliates of the SSRC for their support.

References

- [1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How different are they really? In *SIGMOD 2008*.
- [2] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *FAST 2007*.
- [3] S. Ames, C. Maltzahn, and E. L. Miller. QUASAR: Interaction with file systems using a query and naming language. Technical Report UCSC-SSRC-08-03, University of California, Santa Cruz, September 2008.
- [4] Apple. Spotlight Server: Stop searching, start finding. <http://www.apple.com/server/macosx/features/spotlight/>, 2008.
- [5] S. M. Beitzel, E. C. Jensen, A. Chowdhury, D. Grossman, and O. Frieder. Hourly analysis of a very large topical categorized web query log. In *SIGIR 2004*.
- [6] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuzmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the 19th Symposium on Parallel Algorithms and Architectures (SPAA '07)*, pages 81–92, 2007.
- [7] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
- [8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [9] E. Brewer. *Readings in Database Systems*, chapter Combining Systems and Databases: A Search Engine Retrospective. MIT Press, 4th edition, 2005.
- [10] S. Buttcher and C. L. Clarke. A security model for full-text file system search in multi-user environments. In *FAST 2004*.
- [11] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *SIGMETRICS 1999*.
- [12] Enterprise Strategy Groups. ESG Research Report: storage resource management market on the launch pad, 2007.
- [13] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM ToIS*, 2(4), 1984.
- [14] Fast, A Microsoft Subsidiary. FAST – enterprise search. <http://www.fastsearch.com/>, 2008.
- [15] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files. In *USENIX 1997*.
- [16] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic file systems. In *SOSP 1991*.
- [17] Google, Inc. Google enterprise. <http://www.google.com/enterprise/>, 2008.
- [18] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In *OSDI 1999*.
- [19] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *USENIX Winter 1994*.
- [20] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *FAST 2004*.
- [21] Kazeon. Kazeon: Search the enterprise. <http://www.kazeon.com/>, 2008.
- [22] Kernel.org. inotify official readme. <http://www.kernel.org/pub/linux/kernel/people/rml/inotify/README>, 2008.
- [23] S. Khoshafian, G. Copeland, T. Jagodits, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In *ICDE 1987*.
- [24] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. In *CIKM 2005*.
- [25] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008*.
- [26] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller. High-performance metadata indexing and search in petascale data storage systems. *Journal of Physics: Conference Series*, 125, 2008.
- [27] libkdtree++. <http://libkdtree.alioth.debian.org/>, 2008.
- [28] C. A. Lynch. Selectivity estimation and query optimization in large databases with highly skewed distribution of column values. In *VLDB 1988*.
- [29] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *USENIX Winter 1994*.
- [30] Microsoft, Inc. Enterprise search from microsoft. <http://www.microsoft.com/Enterprisesearch/>, 2008.
- [31] S. Mitra, M. Winslett, and W. W. Hsu. Query-based partitioning of documents and indexes for information lifecycle management. In *SIGMOD 2008*.
- [32] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *USENIX 2006*.
- [33] M. A. Olson. The design and implementation of the Inversion file system. In *USENIX Winter 1993*.
- [34] Oracle. Oracle berkeley db. <http://www.oracle.com/technology/products/berkeley-db/index.html>, 2008.
- [35] J. K. Ousterhout, A. R. Cherenon, F. Douglass, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, Feb. 1988.
- [36] Y. Padioleau and O. Ridoux. A logic file system. In *USENIX 2003*.
- [37] Private Customers. On the efficiency of modern metadata search appliances, 2008.
- [38] J. T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD 1981*.
- [39] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *SOSP 1999*.
- [40] H. A. Simon. On a class of skew distribution functions. *Biometrika*, 42:425–440, 1955.
- [41] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *FAST 2003*.
- [42] M. Stonebraker and U. Cetintemel. "One Size Fits All": An idea whose time has come and gone. In *ICDE 2005*.
- [43] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB 2007*.
- [44] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI 2006*.
- [45] K. Wu, E. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM ToDS*, 31(1), 2006.

Perspective: Semantic data management for the home

Brandon Salmon, Steven W. Schlosser*, Lorrie Faith Cranor, Gregory R. Ganger
*Carnegie Mellon University, *Intel Research Pittsburgh*

Abstract

Perspective is a storage system designed for the home, with the decentralization and flexibility sought by home users and a new semantic filesystem construct, the *view*, to simplify management. A view is a semantic description of a set of files, specified as a query on file attributes, and the ID of the device on which they are stored. By examining and modifying the views associated with a device, a user can identify and control the files stored on it. This approach allows users to reason about what is stored where in the same way (semantic naming) as they navigate their digital content. Thus, in serving as their own administrators, users do not have to deal with a second data organization scheme (hierarchical naming) to perform replica management tasks, such as specifying redundancy to increase reliability and data partitioning to address device capacity exhaustion. Experiences with Perspective deployments and user studies confirm the efficacy of view-based data management.

1 Introduction

Distributed storage is coming home. An increasing number of home and personal electronic devices create, use, and display digitized forms of music, images, videos, as well as more conventional files (e.g., financial records and contact lists). In-home networks enable these devices to communicate, and a variety of device-specific and datatype-specific tools are emerging. The transition to digital homes gives exciting new capabilities to users, but it also makes them responsible for administration tasks usually handled by dedicated professionals in other settings. It is unclear that traditional data management practices will work for “normal people” reluctant to put time into administration.

This paper presents the Perspective distributed filesystem, part of an expedition into this new domain for distributed storage. As with previous expeditions into new computing paradigms, such as distributed operating systems (e.g., [23, 27]) and ubiquitous computing (e.g., [41]), we are building and utilizing a system representing the vision in order to gain experience. In this case, however, the researchers are not representative of the user population. Most will be non-technical people who just want to use the system, but must (begrudgingly)

deal with administration tasks or live with the consequences. Thus, organized user studies will be required as complements to systems experimentation.

Perspective’s design is motivated by a contextual analysis and early deployment experiences [31]. Our interactions with users have made clear the need for decentralization, selective replication, and support for device mobility and dynamic membership. An intriguing lesson is that home users rarely organize and access their data via traditional hierarchical naming—usually, they do so based on data attributes. Computing researchers have long talked about attribute-based data navigation (e.g., semantic filesystems [12, 36]), while continuing to use directory hierarchies. However, users of home and personal storage live it. Popular interfaces (e.g., iTunes, iPhoto, and even drop-down lists of recently-opened Word documents) allow users to navigate file collections via attributes like publisher-provided metadata, extracted keywords, and date/time. Usually, files are still stored in underlying hierarchical file systems, but users often are insulated from naming at that level and are oblivious to where in the namespace given files end up.

Users have readily adopted these higher-level navigation interfaces, leading to a proliferation of semantic data location tools [42, 3, 13, 37, 19]. In contrast, the abstractions provided by filesystems for managing files have remained tightly tied to hierarchical namespaces. For example, most tools require that specific subtrees be identified, by name or by “volumes” containing them, in order to perform *replica management tasks*, such as partitioning data across computers for capacity management or specifying that multiple copies of certain data be kept for reliability. Since home users double as their own system administrators, this disconnect between interface styles (semantic for data access activities and hierarchical for management tasks) naturally creates difficulties.

The Perspective distributed filesystem allows a collection of devices to share storage without requiring a central server. Each device holds a subset of the data and can access data stored on any other (currently connected) device. However, Perspective does not restrict the subset stored on each device to traditional volumes or subtrees. To correct the disconnect between semantic data access and hierarchical replica management, Perspective replaces the traditional volume abstraction with a new

primitive we call a view. A *view* is a compact description of a set of files, expressed much like a search query, and a device on which that data should be stored. For example, one view might be “*all files with type=music and artist=Beatles stored on Liz’s iPod*” and another “*all files with owner=Liz stored on Liz’s laptop*”. Each device participating in Perspective maintains and publishes one or more views to describe the files that it stores. Perspective ensures that any file that matches a view will eventually be stored on the device named in the view.

Since views describe sets of files using the same attribute-based style as users’ other tools, view-based management replica management is easier than hierarchical file management. A user can see what is stored where, in a human-readable fashion, by examining the set of views in the system. She can control replication and data placement by changing the views of one or more devices. Views allow sets of files to overlap and to be described independently of namespace structure, removing the need for users to worry about application-internal file naming decisions or unfortunate volume boundaries. Semantic management can also be useful for local management tasks, such as setting file attributes and security, in addition to replica management. In addition to anecdotal experiences, an extensive lab study of 30 users each performing 10 different management tasks confirms that view-based management is easier for users than volume-based management.

This paper describes view-based management and our Perspective prototype, which combines existing technologies with several new algorithms to implement view-based distributed storage. View-based data placement and view freshness allow Perspective to manage and expose data mobility with views. Distributed update rules allow Perspective to ensure and expose permanence with views (which can be thought of as semantically-defined volumes). Perspective introduces *overlap trees* as a mechanism for reasoning about how many replicas exist of a particular dataset, and where these files are stored, even when no view exactly matches the attributes of the dataset.

Our Perspective prototype is a user-level filesystem which runs on Linux and OS X. In our deployments, Perspective provides normal file storage as well as being the backing store for iTunes and MythTV in one household and in our research environment lounge. Experiments with the Perspective prototype confirm that it can provide consistent, decentralized storage with reasonable performance. Even with its application-level implementation (connected to the OS via FUSE [10]), Perspective performance is within 3% of native filesystem performance for activities of interest.

2 Storage for the home

Storage has become a component of many consumer electronics. Currently, most stored content is captive to individual devices (e.g., DVRs, digital cameras, digital picture frames, and so on), with human-intensive and proprietary interfaces (if any) for copying it to other devices. But, we expect a rapid transition toward exploiting wireless home networking to allow increased sharing of content across devices. Thus, we are exploring how to architect a distributed storage system for the home.

The home is different from an enterprise. Most notably, there are no sysadmins—household members generally deal with administration (or don’t) themselves. The users also interact with their home storage differently, since most of it is for convenience and enjoyment rather than employment. However, much of the data stored in home systems, such as family photos, is both important and irreplaceable, so home storage systems must provide high levels of reliability in spite of lax management practices. Not surprisingly, we believe that home storage’s unique requirements would be best served by a design different than enterprise storage. This section outlines insights gained from studying use of storage in real homes and design features suggested by them.

2.1 What users want

A contextual analysis is an HCI research technique that provides a wealth of in-situ data, perspectives, and real-world anecdotes of the use of technology. It consists of interviews conducted in the context of the environment under study. To better understand home storage, we extensively interviewed all members of eight households (24 people total), in their homes and with all of their storage devices present. We have also gathered experiences from early deployments in real homes. This section lists some guiding insights (with more detailed information available in technical reports [31]).

Decentralized and Dynamic: The users in our study employed a wide variety of computers and devices. While it was not uncommon for them to have a set of primary devices at any given point in time, the set changed rapidly, the boundaries between the devices were porous, and different data was “homed” on different devices with no central server. One household had set up a home server, at one point, but did not re-establish it when they upgraded the machine due to setup complexity.

Money matters: While the cost of storage continues to decrease, our interviews showed that cost remains a critical concern for home users. Note that our studies were conducted well before the Fall 2008 economic crisis. While the same is true of enterprises, home stor-

age rarely has a clear “return on investment,” and the cost is instead balanced against other needs (e.g., new shoes for the kids) or other forms of enjoyment. Thus, users replicate selectively, and many adopted cumbersome data management strategies to save money.

Semantic naming: Most users navigated their data via attribute-based naming schemes provided by their applications, such as iPhoto, iTunes, and the like. Of course, these applications stored the content into files in the underlying hierarchical file system, but users rarely knew where. This disconnect created problems when they needed to make manual copies or configure backup/synchronization tools.

Need to feel in control: Many approaches to manageability in the home tout automation as the answer. While automation is needed, the users expressed a need to understand and sometimes control the decisions being made. For example, only 2 of the 14 users who backed up data used backup tools. The most commonly cited reason was that they did not understand what the tool was doing and, thus, found it more difficult to use the tool than to do the task by hand.

Infrequent, explicit data placement: Only 2 of 24 users had devices on which they regularly placed data in anticipation of needs in the near future. Instead, most users decided on a type of data that belonged on a device (e.g., “all my music” or “files for this semester”) and rarely revisited these decisions, usually only when prompted by environmental changes. Many did regularly copy new files matching each device’s data criteria onto it.

2.2 Designing home storage

From the insights above, we extract guidance that has informed our design of Perspective.

Peer-to-peer architecture: While centralization can be appealing from a system simplicity standpoint, and has been a key feature in many distributed filesystems, it seems to be a non-starter with home users. Not only do many users struggle with the concept of managing a central server, many will be unwilling to invest the money necessary to build a server with sufficient capacity and reliability. We believe that a decentralized, peer-to-peer architecture more cleanly matches the realities we encountered in our contextual analysis.

Single class of replicas: Many previous systems have differentiated between two classes: permanent replicas stored on server devices and temporarily replicas stored on client devices (e.g., to provide mobility) [32, 25]. While this distinction can simplify system design, it introduces extra complexity for users, and prevents users from utilizing the capacity on client devices for reliabil-

ity, which can be important for cost-conscious home consumers. Having only a single replica class removes the client-server distinction from the user’s perception and allows all peers to contribute capacity to reliability.

Semantic naming for management: Using the same type of naming for both data access and management should be much easier for users who serve as their own administrators. Since home storage users have chosen semantic interfaces for data navigation, replica management tools should be adapted accordingly—users should be able to specify replica management policies applied to sets of files identified by semantic naming.

In theory, applications could limit the mismatch by aligning the underlying hierarchy to the application representation. But, this alternative seems untenable, in practice. It would limit the number of attributes that could be handled, lock the data into a representation for a particular application, and force the user to sort data in the way the application desires. Worse, for data shared across applications, vendors would have to agree on a common underlying namespace organization.

Rule-based data placement: Users want to be able to specify file types (e.g., “Jerry’s music files”) that should be stored on particular devices. The system should allow such rules to be expressed by users and enforced by the system as new files are created. In addition to helping users to get the right data onto the right devices, such support will help users to express specific replication rules at the right granularity, to balance their reliability and cost goals.

Transparent automation: Automation can simplify storage management, but many home users (like enterprise sysadmins) insist on understanding and being able to affect the decisions made. By having automation tools use the same flexible semantic naming schemes as users do normally, it should be possible to create interfaces that express human-readable policy descriptions and allow users to understand automated decisions.

3 Perspective architecture

Perspective is a distributed filesystem designed for home users. It is decentralized, enables any device to store and access any data, and allows decisions about what is stored where to be expressed or viewed semantically.

Perspective provides flexible and comprehensible file organization through the use of *views*. A view is a concise description of the data stored on a given device. Each view describes a particular set of data, defined by a semantic query, and a device on which the data is stored. A view-based replica management system guarantees that

any object that matches the view query will eventually be stored on the device named in the view. We will describe our query language in detail in Section 4.1.

Figure 1 illustrates a combination of management tools and storage infrastructure that we envision, with views serving as the connection between the two layers. Users can set policies through management tools, such as those described in Section 5, from any device in the system at any time. Tools implement these changes by manipulating views, and the underlying infrastructure (Perspective) in turn enforces those policies by keeping files in sync among the devices according to the views. Views provide a clear division point between tools that allow users to manage data replicas and the underlying filesystem that implements the policies.

View-based management enables the design points outlined in Section 2.2. Views provide a primitive allowing users to specify meaningful rule-based placement policies. Because views are semantic, they unify the naming used for data access and data management. Views are also defined in a human-understandable fashion, providing a basis for transparent automation. Perspective provides data reliability using views without restricting their flexibility, allowing it to use a single replica class.

3.1 Placing file replicas

In Perspective, the views control the distribution of data among the devices in the system. When a file is created or updated, Perspective checks the attributes of the file against the current list of views in the system and sends an update message to each device with a view that contains that file. Each device can then independently pull a copy of the update.

When a device, A, receives an update message from another device, B, it checks that the updated file does, indeed, match one or more views that A has registered. If the file does match, then A applies the update from B. If there is no match, which can occur if the attributes of a file are updated such that it is no longer covered by a view, then A ensures that there is no replica of the file stored locally.

This simple protocol automatically places new files, and also keeps current files up to date according to the current views in the system. Simple rules described in Section 4.3 assure that files are never dropped due to view changes.

Each device is represented by a file in the filesystem that describes the device and its characteristics. Views themselves are also represented by files. Each device registers a view for all device and view files to assure they are replicated on all participating devices. This allows appli-

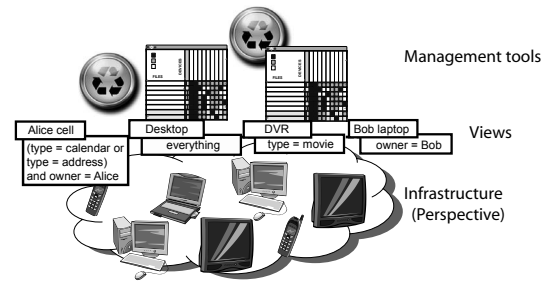


Figure 1: View-based architecture. Views are the interface between management tools and the underlying heterogeneous, disconnected infrastructure. By manipulating the views, tools can specify data policies that are then enforced by the filesystem.

cations to manage views through the standard filesystem interfaces, even if not all devices are currently present.

3.2 View-based data management

This subsection presents three scenarios to illustrate view-based management.

Traveling: Harry is visiting Sally at her house and would like to play a new U2 album for her while he is at her house. Before leaving, he checks the views defined on his wireless music player and notices the songs are not stored on the device, though he can play them from his laptop where they are currently stored. He asks the music player to pull a copy of all U2 songs, which the player does by creating a new view for this data. When the synchronization is complete, the filesystem marks the view as complete, and the music player informs Harry.

He takes the music player over to Sally's house. Because the views on his music player are defined only for his household and the views on Sally's devices for her household, no files are synchronized. But, queries for "all music" initiated from Sally's digital stereo can see the music files on Harry's music player, so they can listen to the new U2 album off of Harry's music player on the nice stereo speakers, while he is visiting.

Crash: Mike's young nephew Oliver accidentally pushes the family desktop off of the desk onto the floor and breaks it. Mike and his wife Carol have each configured the system to store their files both on their respective laptop and on the desktop, so their data is safe. When they set up the replacement computer, a setup tool pulls the device objects and views from other household devices. The setup tool gives them the option to replace an old device with this computer, and they choose the old desktop from the list of devices. The tool then creates views on the device that match the views on the old desktop and deletes the device object for the old computer. The data from Mike and Carol's laptops is transferred to the new

desktop in the background over the weekend.

Short on space: Marge is working on a project for work on her laptop in the house. While she is working, a capacity automation tool on her laptop alerts her that the laptop is short on space. It recommends that files created over two years ago be moved to the family desktop, which has spare space. Marge, who is busy with her project, decides to allow the capacity tool to make the change. She later decides to keep her older files on the external hard drive instead, and makes the change using a view-editing interface on the desktop.

4 Perspective design

This section details three aspects of Perspective: semantic search and naming, consistent partial replication of sets of files, and reliability maintenance and reasoning.

The Perspective prototype is implemented in C++ and runs at user-level using FUSE [10] to connect with the system. It currently runs on both Linux and Macintosh OS X. Perspective stores file data in files in a repository on the machine's local filesystem and metadata in a SQLite database with an XML wrapper. Our prototype implements all of the features described in this paper except garbage collection and some advanced update log features.

The prototype system has supporting one researcher's household's DVR, which is under heavy use; it is the exclusive television for him and his four roommates, and is also frequently used by sixteen other friends in the same complex. It has also stored one researcher's personal data for about a year. It has also been the backing store for the DVR in the lounge for our research group for several months. We are preparing the system for deployment in several non-technical households for a wider, long-term user study over several months.

4.1 Search and naming

All naming in Perspective uses semantic metadata. Therefore, search is a very common operation both for users and for many system operations. Metadata queries can be made from any device and Perspective will return references to all matching files on devices currently accessible (e.g., on the local subnet), which we will call the current *device ensemble* [33]. Views allow Perspective to route queries to devices containing all needed files and, when other devices suffice, avoid sending queries to power-limited devices. While specialized applications may use the Perspective API directly, we expect most applications to access files through the standard VFS layer, just as they access other filesystems. Perspective pro-

vides this access using *front ends* that support a variety of user-facing naming schemes. These names are then converted to Perspective searches, which are then passed on to the filesystem. Our current prototype system implements four front ends that each support a different organization: directory hierarchies, faceted metadata, simple search, and hierarchies synthesized from the values of specific tags.

Query language and operations: We use a query language based on a subset of the XPath language used for querying XML. Our language includes logic for comparing attributes to literal values with equality, standard mathematical operators, string search, and an operator to determine if a document contains a given attribute. Clauses can be combined with the logical operators *and*, *or*, and *not*. Each attribute is allowed to have a single value, but multi-value attributes can be expressed in terms of single value attributes, if necessary. We require all comparisons to be between attributes and constant values.

In addition to standard queries, we support two operations needed for efficient naming and reliability analysis. The first is the *enumerate values* query, which returns all values of an attribute found in files matching a given query. The second is the *enumerate attributes* query, which returns all the unique attributes found in files matching a given query. These operations must be efficient; fortunately we can support them at the database level using indices, which negate the need for full enumeration of the files matching the query.

This language is expressive enough to capture many common data organization schemes (e.g., directories, unions [27], faceted metadata [43], and keyword search) but is still simple enough to allow Perspective to efficiently reason about the overlap of queries. Perspective can support any of the replica management functions described in this paper for any naming scheme that can be converted into this language.

Overlap evaluation is commonly used to compare two queries. The overlap evaluation operation returns one of three values when applied to two queries: one query *subsumes* the other, the two queries have *no-overlap*, or the relationship between them is *unknown*. Note that the comparison operator is used for efficiency but not correctness, allowing for a trade-off between language complexity and efficiency. For example, Perspective can determine that the query *all files where date < January, 2008* is subsumed by the query *all files where date < June, 2008*, and that the query *all files where owner=Brandon* does not overlap with the query *all files where owner=Greg*. However, it cannot determine the relationship between the queries *all files where*

type=Music and *all files where album=The Joshua Tree*. Perspective will correctly handle operations on the latter two queries, but at some cost in efficiency.

4.2 Partial replication

Perspective supports partial replication among the devices in a home. Devices in Perspective can each store disjoint sets of files — there is no requirement that any master device store all files or that any device mirror another in order to maintain reliability. Previous systems have supported either partial replication [16, 32] or topology independence [40], but not both. PRACTI [7] provided a combination of the two properties tied to directories, but probably could be extended to work in the semantic case. Recently, Cimbiosis [28] has also provided partial replication with effective topology independence, although it requires all files to be stored on some master device. We present Perspective’s algorithms to show that it is possible to build a simple, efficient consistency protocol for a view-based system, but a full comparison with previous techniques is beyond the scope of this paper. The related work section presents the differences and similarities with previous work.

Synchronization: Devices that are not currently accessible at the time of an update will receive that update at synchronization time, when the two devices exchange information about updates that they may have missed. Device and view files are always synchronized before other files, to make sure the device does not miss files matching new views. Perspective employs a modified update log to limit the exchanges to only the needed information, much like the approach used in Bayou [40]. However, the flexibility of views makes this task more challenging.

For each update, the log contains the metadata for the file both before and after the update. Upon receiving a sync request, a device returns all updates that match the views for the calling device either before or after the update. As in Bayou, the update log is only an optimization; we can always fall back on full file-by-file synchronization.

Conventional full synchronization can be problematic for heterogeneous devices with partial replication, especially for resource- and battery-limited devices. For example, if a cell phone syncs with a desktop computer, it is not feasible for the cell phone to process all of the files on the desktop, even occasionally. To address this problem, Perspective includes a second synchronization option. Continuing the example, the cell phone first asks the desktop how many updates it would return. If this number is too large, the cell phone can pass the metadata of all the files it owns to the desktop, along with the view query, and ask the desktop for updates for any files that match the view or are contained in the set of files currently on the

cell phone. At each synchronization, the calling device can choose either of these two methods, reducing synchronization costs to $O(N_{smaller})$, where $N_{smaller}$ is the number of files stored on the smaller device.

Full synchronizations will only return the most recent version of a file, which may cause gaps in the update logs. If the update log has a gap in the updates for a file, recognizable by a gap in the versions of the before and after metadata, the calling device must pass this update back to other devices on synchronization even if the metadata does not match the caller’s views, to avoid missing updates to files which used to match a view, but now do not.

Consistency: As with many file systems that support some form of eventual consistency, Perspective uses version vectors and epidemic propagation to ensure that all file replicas eventually converge to the same version. Version vectors in Perspective are similar to those used in many systems; the vector contains a version for each replica that has been modified. Because Perspective does not have the concept of volumes, it does not support volume-level consistency like Bayou. Instead, it supports file-level consistency, like FICUS [16].

To keep all file replicas consistent, we need to assure that updates will eventually reach all replicas. If all devices in the household sync with one another occasionally, this property will be assured. While this is a reasonable assumption in many homes, we do not require full pairwise device synchronization. Like many systems built on epidemic propagation, a variety of configurations satisfy this property. For example, even if some remote device (e.g., a work computer) never returns home, the property will still hold as long as some other device that syncs with the remote device and does return home (e.g., a laptop) contains all the data stored on the remote device. System tools might even create views on such devices to facilitate such data transfer, similar to the routing done in Footloose [24]. Alternately, a sync tree, as that used in Cimbiosis [28] could be layered on top of Perspective to provide connectedness guarantees.

By tracking the timestamps of the latest complete sync operation for each device in the household, devices provide a *freshness timestamp* for each view. Perspective can guarantee that all file versions created before the freshness timestamp for a view are stored on that view’s device. It can also recommend sync operations needed to advance the freshness timestamp for any view.

Conflicts: Any system that supports disconnected operation must deal with *conflicts*, where two devices modify the same file without knowledge of the other device’s modification. We resolve conflicts first with a *pre-resolver*, which uses the metadata of the two versions to

deterministically choose a winning and losing version. Our pre-resolver can be run on any device without any global agreement. It uses the file's modification time and then the sorted version vector in the case of a tie. But, instead of eliminating the losing version, the pre-resolver creates a new file, places the losing version in this new file. It then tags the new file with all metadata from the losing version, as well as tags marking it as a conflict file and tying it to the winning version. Later, a *full resolver*, which may ask for user input or use more sophisticated logic, can search for conflict objects, remove duplicates, and adjust the resolution as desired.

Capacity management: Pushing updates to other devices can be problematic if those devices are at full capacity. In this case, the full device will refuse subsequent updates, and mark the device file noting that the device is out of space. Until a user or tool corrects the problem, the device will continue to refuse updates, although other devices will be able to continue. However, management tools built on top of Perspective should help users address capacity problems before they arise.

File deletion: As in many other distributed filesystems, when a file is removed, Perspective keeps a *tombstone* marker that assures all replicas of the file in the system are deleted, but is ignored by all naming operations. Perspective uses a two-phase garbage collection mechanism, like that used in FICUS, between all devices with views that match the file to which the tombstone belongs. Note that deletion of a file removes all replicas of a file in the system, which is a different operation from dropping a particular replica of a file (done by manipulating views). This distinction also exists in any distributed filesystem allowing replication.

View and device objects: Each device is only required to store view and device objects from devices that contain replicas of files it stores, although they must also temporarily store view and device files for devices in the current ensemble in order to access their files. Because views are very small (hundreds of bytes), this is tractable, even for small devices like cell phones.

4.3 Reliability with partial replication

In order to manage data semantically, users must be able to provide fault-tolerance on data split semantically across a distributed set of disconnected, eventually-consistent devices. Perspective enables semantic fault-tolerance through two new algorithms, and provides a way to efficiently reason about the number of replicas of arbitrary sets of files. It also assures that data is never lost despite arbitrary and disconnected view manipulation using three simple distributed update rules.

Reasoning about number of replicas: Reasoning about the reliability of a storage system — put simply, determining the level of replication for each data item — is a challenge in a partially-replicated filesystem. Since devices can store arbitrary subsets of the data, there are no simple rules that allow all of the replicas to be counted. A naïve solution would be to enumerate all of the files on each device and count replicas. Unfortunately, this would be prohibitively expensive and would be possible only if all devices are currently accessible.

Fortunately, Perspective's views compactly and fully describe the location of files in terms of their attributes. Since there are far fewer views than there are file replicas in the system, it is cheaper to reason about the number of times a particular query is replicated among all of the views in the system than to enumerate all replicas. The files in question could be replicated exactly (e.g., all of the family's pictures are on two devices), they could be subsumed by multiple views (e.g., all files are on the desktop and all pictures are on the laptop), or they could be replicated in part on multiple devices but never in full on any one device (e.g., Alice's pictures are on her laptop and desktop, while Bob's pictures are on his laptop and desktop — among all devices, the entire family's pictures have two replicas).

To efficiently reason about how views overlap, Perspective uses *overlap trees*. An overlap tree encapsulates the location of general subsets of data in the system, and thus simplifies the task of determining the location of the many data groupings needed by management tools. An overlap tree is currently created each time a management application starts, and then used throughout the application's runtime to answer needed overlap queries.

Overlap trees are created using the enumeration queries described in Section 4.1. Each node contains a query, that describes the set of data the node represents. Each leaf node represents a subset of files whose location can be precisely quantify using the views and records the devices that store that subset. Each interior node of the tree encodes a subdivision of the attribute space, and contains a list of child nodes, each of which represents a subset of the files that the parent node represents. We begin building the tree by enumerating all of the attributes that are used in the views found in the household.

We create a root node for the tree to represent all files, choose the first attribute in our attribute list, and use the enumerate values query to find all values of this attribute for the current node's query. We then create a child node from each value with a query of the form *<parent query> and attribute=value*. We compare the query for each child node against the complete views on all devices. If the compare operator can give a precise answer

(i.e., not *unknown*) for whether the query for this node is stored on each device in the home, then this node is a leaf and we can stop dividing. Otherwise, we recursively perform this operation on the child node, dividing it by the next attribute in our list. Figure 2 shows an example overlap tree. The ordering of the attribute list could be optimized to improve performance of the overlap tree, but in the current implementation we leave it unordered.

When we create an overlap tree, we may not have all the information needed to construct the tree. For example, if we currently only have access to Brian’s files, we may incorrectly assume that all music files in the household are owned by Brian, when music files owned by Mary exist elsewhere in the system. The tree construction mechanism makes a notation in a node if it cannot guarantee that all matching files are available via the views. When checking for overlaps, if a marked node is required the tree will return an *unknown* value, but it will still correctly compute overlaps for queries that do not require these nodes. To avoid this restriction, devices are free to cache and update an overlap tree, rather than recreating the overlap tree when each management application starts. The tree is small, making caching it easy. To keep it up to date, a device can publish a view for all files, and then use the updates to keep the cached tree up to date.

Once we have constructed the overlap tree, we can use it to determine the location and number of full copies in the system of the files for any given query. Because the tree caches much of the overlap processing, each individual query request can be processed efficiently. We do so by traversing all of the leaf nodes and finding those that overlap with the given view or query. We may occasionally need to perform more costly overlap detection, if the attribute in a leaf node does not match any of the attributes in the query. For example, in the overlap tree in Figure 2, if we were checking to see if the query *album=Joshua Tree* was contained in the node *owner=Mary and type=Music* we would use the enumerate values query to determine the values of “type” for the query *album=Joshua Tree and owner=Mary*. If “Music” is the only value, then we can count this node as a full match in our computations. Otherwise, we cannot. This extra comparison is only valid if we can determine via the views that all files in the query for which we are computing overlaps are accessible

Attributes with larger cardinalities can be handled more efficiently by selectively expanding the tree. For example, if a view is defined on *date* $< T$, we need only expand the attribute date into three sub-nodes, one for *date* $< T$, one for *date* $\geq T$, and one for *has no date attribute*.

Note that the number of attributes used in views at any one time is likely to be much smaller than the total num-

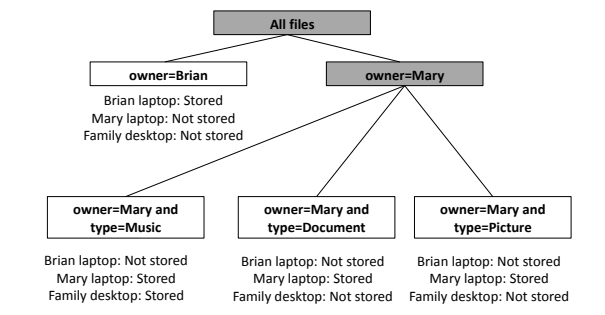


Figure 2: Overlap tree. This figure shows an example overlap tree, constructed from a three-device, three-view scenerio: Brian’s files stored on Brian’s laptop, Mary’s files stored on Mary’s laptop, and Mary’s music stored on the Family desktop. Shaded nodes are interior nodes, unshaded nodes are leaf nodes. Each leaf node lists whether this query is stored on each device the household.

ber of attributes in the system, and both of these will be much smaller than the total number of files or replicas. For example, in our contextual analysis, most households described settings requiring around 20 views and 5 attributes. None of households we interviewed described more than 30 views, or more than 7 attributes. Because the number of relevant attributes is small, overlap tree computations are fast enough to allow us to compute them in real time as the user browses files. We will present a performance comparison of overlap trees to the naïve approach in Section 6.

Update rules: Perspective maintains permanence by guaranteeing that files will never be lost by changes to views or addition or removal of devices, regardless of the order, timing, or origin of the changes, freeing the user from worrying about these problems when making view changes. We also provide a guarantee that, once a version of a file is stored on the devices associated with all overlapping views, it will always be stored in all overlapping views, which provides a strong assurance on the number of copies in the system based on the current views. View freshness timestamps, as described in Section 4.2, allow Perspective to guarantee that all updates created before a given timestamp are safely stored in the correct locations, and thus have the fault-tolerance implied by the views. These guarantees are assured by careful adherence to three simple rules: 1) When a file replica is modified by a device, it is marked as “modified.” Devices cannot evict modified replicas. Once a modified replica has been pulled by a device holding a view covering it, the file can be marked as unmodified and then removed. 2) A newly created view cannot be considered complete until it has synced with all devices with overlapping views or synced with one device with a view that subsumes the new view. 3) When a view is removed, all replicas in it are marked as modified. The replicas are then removed when they conform to rule 1.

These rules ensure that devices will not evict modified replicas until they are safely on some “stable” location (i.e., in a completely created view). The rules also assure that a device will not drop a file until it has confirmed that another up-to-date replica of the file exists somewhere in the system. However, a user can force the system to drop a file replica without assuring another replica exists, if she is confident that another replica exists and is willing to forgo this system protection. With these rules, Perspective can provide permanence guarantees without requiring central control or limiting when or where views can be changed.

4.4 Security and cross-household sharing

Security is not a focus in this paper, but is certainly a concern for users and system designers alike. While Perspective does not currently support it, we envision using mechanisms such as those promoted by the UIA project [8]. Our current prototype supports voluntary access control using simplified access control lists. While all devices are able to communicate and share replicas with one another, even aside from security concerns it is helpful to divide households from one another to divide management and view specification. To do so, Perspective maintains a household ID for each device and each file. Views are specified on files within the given household, to avoid undesired cross-syncing. However, the fundamental architecture of Perspective places no restrictions on how these divisions are made.

5 View manager interface

To explore view-based management, we built a *view manager* tool to allow users to manipulate views.

Customizable faceted metadata: One way of visualizing and accessing semantic data is through the use of *faceted metadata* [43]. Faceted metadata allows a user to choose a first attribute to use to divide the data and a value at which to divide. Then, the user can choose another attribute to divide on, and so on. Faceted metadata helps users browse semantic information by giving them the flexibility to divide the data as needed. But, it can present the user with a dizzying array of choices in environments with large numbers of attributes.

To curb this problem, we developed *customizable faceted metadata* (CFM), which exposes a small user-selected set of attributes as directories plus one additional *other groupings* directory that contains a full list of possible attributes. The user can customize which attributes are displayed in the original list by moving folders between the base directory and the *other groupings* directory. These

preferences are saved in a customization object in the filesystem. The file structure on the left side of the interface in Figure 3 illustrates CFM. Perspective exposes CFM through the VFS layer, so it can be accessed in the same way as a normal hierarchical filesystem.

View manager interface: The view manager interface (Figure 3), allows users to create and delete views on devices and to see the effects of these actions. This GUI is built in Java and makes calls into the view library of the underlying filesystem.

The GUI is built on Expandable Grids [29], a user interface concept initially developed to allow users to view and edit file system permissions. Each row in the grid represents a file or file group, and each column represents a device in the household. The color of a square represents whether the files in the row are stored on the device in the column. The files can be “all stored” on the device, “some stored” on the device, or “not stored” on the device. Each option is represented by a different color in the square. By clicking on a square a user can add or remove the given files from the given device. Similarly to file permissions, this allows users to manipulate actual storage decisions, instead of rule lists.

An extra column, labeled “Summary of failure protection,” shows whether the given set of files is protected from one failure or not, which is true if there are at least two copies of each file in the set. By clicking on an unbacked-up square, the user can ask the system to assure that two copies of the files are stored in the system, which it will do by placing any extra needed replicas on devices with free space.

An extra row contains all unique views and where they are stored, allowing a user to see precisely what data is stored on each device at a glance.

6 Evaluation

Our experience from working with many home storage users suggests that users are very concerned about the time and effort spent managing their devices and data at home, which has motivated our design of Perspective, as well as our evaluation. Therefore, we focus our study primarily on the usability of Perspective’s management capabilities and secondarily on its performance overhead.

We conducted a lab study in which non-technical users used Perspective, outfitted with appropriate user interfaces, to perform home data management tasks. We measured accuracy and completion time of each task. In order to insulate our results as much as possible from the particulars of the user interface used for each primitive, we built similar user interfaces for each primitive using

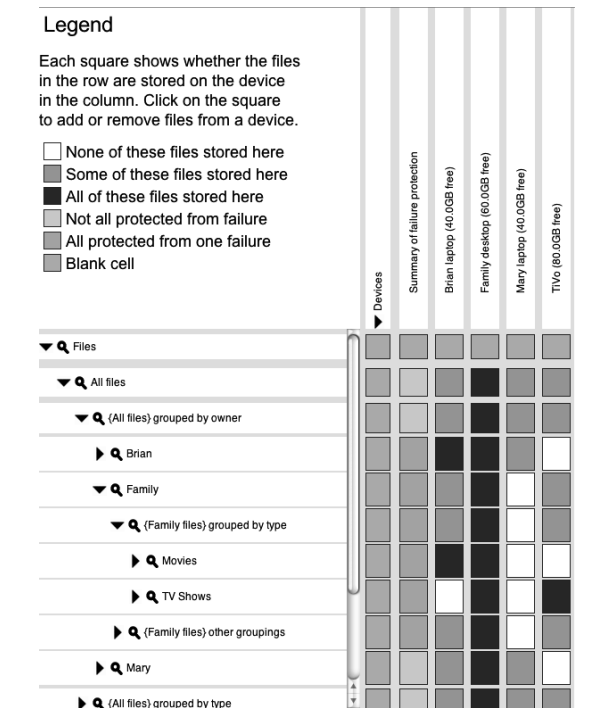


Figure 3: View manager interface. A screen shot of the view manager GUI. On the left are files, grouped using faceted metadata. Across the top are devices. Each square shows whether the files in the row are stored on the device in the column.

the Expandable Grids UI toolkit [29].

Views-facet interface: The views-facet interface was described in Section 5. It uses CFM to describe data, and allows users to place any set of data described by the faceted metadata on any device in the home.

Volumes interface: This user interface represents a similar interface built on top of a more conventional volume-based system with directory hierarchies. Each device is classified as a client or server, and this distinction is listed in the column along with the device name. The volumes abstraction only allows permanent copies of data to be placed on servers, and it restricts server placement policies on volume boundaries. We defined each root level directory (based on user) as a volume. The abstraction allows placement of a copy of any subtree of the data on any client device, but these replicas are only temporary caches and are not guaranteed to be permanent or complete. The interface distinguishes between temporary and permanent replicas by color. The legend displays a summary of the rules for servers and permanent data and for clients and temporary data.

Views-directory interface: To tease apart the effects of semantic naming and using a single replica class, we evaluated an intermediate interface, which replaces the CFM organization with a traditional directory hierarchy.

Otherwise, it is identical to the views-facet interface. In particular, it allows users to place any subtree of the hierarchy on any device.

6.1 Experiment design

Our user pool consisted of students and staff from nearby universities in non-technical fields who stated that they did not use their computers for programming. We did a *between-group* comparison, with each participant using one of the three interfaces described above. We tested 10 users in each group, for a total of 30 users overall. The users performed a *think-aloud* study in which they spoke out loud about their current thoughts and read out loud any text they read on the screen, which provides insight into the difficulty of tasks and users' interpretation. All tasks were performed in a *latin square* configuration, which guarantees that every task occurs in each position in the ordering, and each task is equally likely to follow any other task.

We created a filesystem with just over 3,000 files, based on observations from our contextual analysis. We created a setup with two users, Mary and Brian, and a third "Family" user with some shared files. We modeled Brian's file layout on the Windows music and pictures tools and Mary's on Apple's iTunes and iPhoto file trees. Our setup included four devices: two laptops, a desktop, and a DVR. We also provided the user with iTunes and iPhoto, with the libraries filled with all of the matching data from the filesystem. This allowed us to evaluate how users convert from structures in the applications to the underlying filesystem.

6.2 Tasks

Each participant performed the same set of tasks, which we designed based on our contextual analysis. We started each user with a 5 to 10 minute training task, after which our participants performed 10 data management tasks. While space constraints preclude us including the full text for all of them, as we discuss each class of tasks, we include the text of one example task. For this study, we chose tasks to illustrate the differences between the approaches. A base-case task that was similar in all interfaces confirmed that, on these similar tasks, all interfaces performed similarly. The tasks were divided into two types: single replica tasks, and data organization tasks.

Single replica tasks: Two single replica tasks (LH and CB) required the user to deal with distinctions between permanent and temporary replicas to be successful.

Example task, Mary's laptop comes home (LH): "*Mary has not taken her laptop on a trip with her for a while now, so she has decided to leave it in the house and make*

an extra copy of her files on it, in case the Family desktop fails. However, Brian has asked her not to make extra copies of his files or of the Family files. Make sure Mary's files are safely stored on her laptop."

Mary's laptop was initially a client in the volume case. This task asked the user to change it to a server before storing data there. This step was not required for the single replica class interfaces, as all devices are equivalent.

Note that because server/client systems, unlike Perspective, are designed around non-portable servers for simplicity, it is not feasible to simply make all devices servers. Indeed, the volume interface actually makes this task much simpler than current systems; in the volume interface, we allow the user to switch a device from server to client using a single menu option, where current distributed filesystems require an offline device reformat.

Data organization tasks: The data organization tasks required users to convert from structures in the iTunes and iPhoto applications into the appropriate structures in the filesystem. This allowed us to test the differences between a hierarchical and semantic, faceted systems. The data organization tasks are divided into three types: aggregation, comprehension, and sparse collection tasks.

Aggregation: One major difference between semantic and hierarchical systems is that because the hierarchy forces a single tree, tasks that do not match the current tree require the user to aggregate data from multiple directories. This is a natural case as homes fill with aggregation devices and data is shared across users and devices. However, in a hierarchical system, it is difficult for users to know all the folders that correspond to a given application grouping. Users often erroneously assumed all the files for a given collection were in the same folder. The semantic structure mitigates this problem, since the user is free to use a filesystem grouping suited to the current specific task.

Example task, U2 (U2): *"Mary and Brian share music at home. However, when Mary is on trips, she finds that she can't listen to all the songs by U2 on her laptop. She doesn't listen to any other music and doesn't want other songs taking up space on her laptop, but she does want to be able to listen to U2. Make sure she can listen to all music by the artist U2 on her trips."*

As may often be the case in the home, the U2 files were spread across all three user's trees in the hierarchical interfaces. The user needed to use iTunes to locate the various folders. The semantic system allowed the user to view all U2 files in a single grouping.

Aggregation is also needed when applications sort data differently from what is needed for the current task. For example, iPhoto places modified photos in a separate

folder tree from originals, making it tricky for users to get all the files for a particular event. The semantic structure allows applications to set and use attributes, while allowing the user to group data as desired.

Example task, Rafting (RF): *"Mary and Brian went on a rafting trip and took a number of photos, which Mary remembers they labeled as 'Rafting 2007'. She wants to show her mother these photos on Mary's laptop. However, she doesn't want to take up space on her laptop for files other than the 'Rafting 2007' files. Make sure Mary can show the photos to her mother during her visit."*

The rafting photos were initially in Brian's files, but iPhoto places modified copies of photos in a separate directory in the iPhoto tree. To find both folders, the user needed to explore the group in iPhoto. The semantic system allows iPhoto to make the distinction, while allowing the user to group all files from this roll in one grouping.

Comprehension: Applications can allow users to set policies on application groupings, and then convert them into the underlying hierarchy. However, in addition to requiring multiple implementations and methods for the same system tasks, this leads to extremely messy underlying policies, which make it difficult for users to understand, especially when viewing it from another application. In contrast, semantic systems can retain a description of the policy as specified by the application, making them easier for users to understand.

Example task, Traveling Brian (TB): *"Brian is taking a trip with his laptop. What data will he be able to access while on his trip? You should summarize your answer into two classes of data."*

Brian's laptop contained all of his files and all of the music files in the household. However, because iTunes places TV shows in the Music repository, the settings included all of the music subfolders, but not the "TV Shows" subfolder, causing confusion. In contrast, the semantic system allows the user to specify both of these policies in a single view, while still allowing applications to sort the data as needed.

Note that this particular task would be simpler if iTunes chose to sort its files differently, but the current iTunes organization is critical for other administrative tasks, such as backing up a user's full iTunes library. It is impossible to find a single hierarchical grouping that will be suited to all needed operations. This task illustrates how these kinds of mismatches occur even for common tasks and well-behaved applications.

Sparse collection: Two sparse collection tasks (BF and HV) required users to make policies on collections that contain single files from across the tree, such as song playlists. These structures do not lend themselves well

to a hierarchical structure, so they are kept externally in application structures, forcing users to re-create these policies by hand. In contrast, semantic structures allow applications to push these groupings into the filesystem.

Example task, Brian favorites (BF): *“Brian is taking a trip with his laptop. He doesn’t want to copy all music onto his laptop as he is short on space, but he wants to have all of the songs on the playlist “Brian favorites”.”*

Because the playlist does not exist in the hierarchy, the user had to add the nine files in the playlist individually, after looking up the locations using iTunes. In the semantic system, the playlist is included as a tag, allowing the user to specify the policy in a single step.

6.3 Results

All of the statistically significant comparisons are in favor of the facet interface over the alternative approaches, showing the clear advantage of semantic management for these tasks. For the single replica tasks the facet and directory interfaces perform comparably, as expected, with an average accuracy of 95% and 100% respectively, compared to an average of 15% for the volume interface. For the data organization tasks, the facet interface outperforms the directory and volume interfaces with an average accuracy of 66% compared to 14% and 6% respectively. Finally, while the accuracy of sparse tasks is not significantly different, the average time for completion for the facet interface is 73 seconds, compared to 428 seconds for the directory interface and 559 seconds for the volume interface. We discuss our statistical comparisons and the tasks in more detail in this section.

Statistical analysis: We performed a statistical analysis on our accuracy results in order to test the strength of our findings. Because our data was not well-fitted to the chi-squared test, we used a one-sided Fisher’s Exact Test for accuracy and a t-test to compare times. We used Benjamini-Hochberg correction to adjust our p values to correct for our use of multiple comparisons. As is conventional in HCI studies, we used $\alpha = .05$. All the comparisons mentioned in this section were statistically significant, except where explicitly mentioned.

Single replica tasks: Figure 4 shows results from the single replica tasks. As expected, the directory and view interfaces, which both have a single replica class, perform equivalently, while the volume interface suffers heavily due to the extra complexity of two distinct replica classes. The comparisons between the single replica interfaces and the volume interface are all statistically significant. We do not show times, because they showed no appreciable differences.

Data organization tasks: Results from the three aggre-

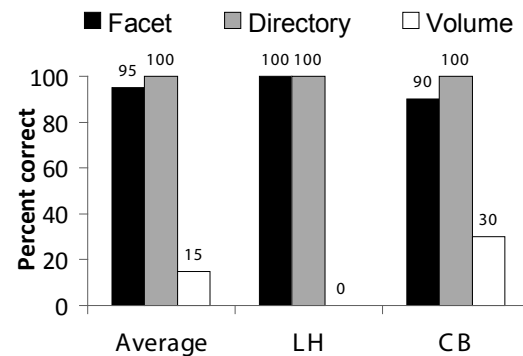


Figure 4: Single replica task results. This graph shows the results of the single replica tasks.

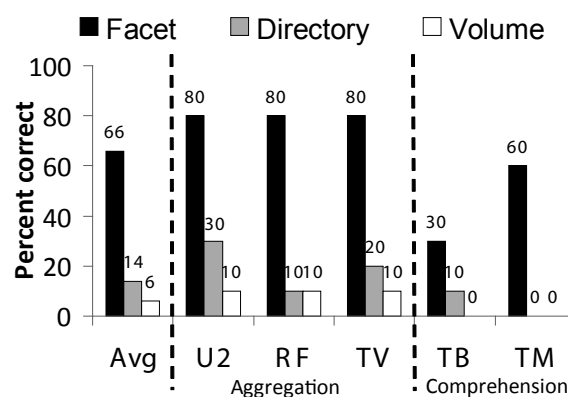


Figure 5: Data organization task results. This graph shows the results from the aggregation and comprehension tasks.

gation tasks (U2, RF, and TV), and the two comprehension tasks (TB and TM) are shown in Figure 5. As expected, the faceted metadata approach performs significantly better than the alternative approaches, as the filesystem structure more closely matches that of the applications. The facet interface is statistically better than both the other interfaces in the aggregation tasks, but we would need more data for statistical significance for the comprehension tasks.

Figure 6 shows the accuracy and time metrics for the sparse tasks (BF and HV). Note that none of the accuracy comparisons are statistically significant. This is because in the sparse tasks, each file is in a unique location, making the correlation between application structure and filesystem structure clear, but very inconvenient. In contrast, for the other aggregation tasks the correlation between application and structures and the filesystem was hazy, leading to errors. However, setting the policy on each individual file was extremely time consuming, leading to a statistically significant difference in times. The one exception is the HV task, where too few volume

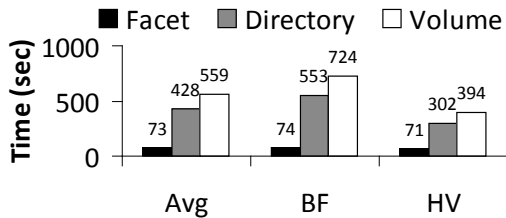


Figure 6: Sparse collection task results. This graph shows the results from all of the sparse collection tasks.

users correctly performed the task to allow comparison with the other interfaces. Indeed, the hierarchical interfaces took an order of magnitude longer than the facet interface for these tasks. Thus re-creating the groups was difficult, leading to frustration and frequent grumbling that “there must be a better way to do this.”

6.4 Performance evaluation

We have found that Perspective generally incurs negligible overhead over the base filesystem, and its performance is sufficient for everyday use. Using overlap trees to reason about the location of files based on the available views is a significant improvement over simpler schemes. All our tests were run on a MacBook Pro 2.5GHz Intel Core Duo with 2GB RAM running Macintosh OS X 10.5.4.

Performance overhead: Our benchmark writes 200 4MB files, clearing the cache by writing a large amount of data elsewhere and then re-reading all 800MB. This sequential workload on small files simulates common media workloads. For these tasks, we compared Perspective to HFS+, the standard OS X filesystem. Writing the files on HFS+ and Perspective took 18.1 s and 18.6 s, respectively. Reading them took 17.0 s and 17.2 s, respectively. Perspective has less than a 3% overhead in both phases of this benchmark. In a more real-world scenario, Perspective has been used by the authors for several months as the backing store for several multi-tuner DVRs, without performance problems.

Overlap trees: Overlap trees allow us to efficiently compute how many copies of a given file set are stored in the system, despite the more flexible storage policies that views provide. It is important to make this operation efficient because, while it is only used in administration tasks, these tasks require calculation of a large number of these overlaps in real time as the user browses and manipulates data placement policies.

Figure 7 summarizes the benefits of overlap trees. We compared overlap trees to a simple method that enumerates all matching files and compares them against the

Num files	Create OT	OT no probe	OT w/ probe	Simple
100	9.6ms	0.3ms	3.5ms	961ms (.9sec)
1000	29ms	0.6ms	3.8ms	12759ms (12sec)
10000	249ms	0.6ms	3.4ms	95049ms (95sec)

Figure 7: Overlap tree benchmark. This table shows the results from the overlap tree benchmark. It compares the time to create a tree and perform an overlap comparison, with or without probes, and compares to the simple enumerate approach. The results are the average of 10 runs.

views in the system. We break out the cost for tree creation and then the cost to compute an overlap. The “probe” case uses a query and view set that requires the overlap tree to probe the filesystem to compute the overlap, while the “no probe” case can be determined solely through query comparisons. Overlap trees take a task that would require seconds or minutes and turns it into a task requiring milliseconds.

7 Related work

A primary contribution of Perspective is the use of semantic queries to *manage the replication of data*. Specifically, it allows the system to provide accessibility and reliability guarantees over semantic, partially replicated data. This builds on previous semantic systems that used queries to *locate* data, and hierarchies to manage data. Our user study evaluation shows that, by supporting semantic management, Perspective can simplify important management tasks for end users.

Another contribution is a filesystem design based on in-situ analysis of the home environment. This overall design could be implemented on top of a variety of underlying filesystem implementations, but we believe that a fully view-based system provides simplicity to both user and designer by keeping the primitives similar throughout the system. While no current system provides all of the features of Perspective, Perspective builds on a wealth of previous work in data placement, consistency, search and publish/subscribe event notification. In this section we discuss this related work.

Data placement: Views allow flexible data placement used to provide both reliability and mobility. Views are another step in a long progression of increasingly flexible data placement schemes.

The most basic approach to storing data in the home is to put all of the data on a single server and make all other devices in the home clients of this server. Variations of this approach centralize control, while allowing data to be cached on devices [18, 35].

To provide better reliability, AFS [34] expanded the single server model to include a tier of replicated servers,

each connected in a peer-to-peer fashion. However, clients cannot access data when they are out of contact with the servers. Coda [32] addressed this problem by allowing devices to enter a disconnected mode, in which devices use locally cached data defined by user hoarding priorities. However, hoarded replicas do not provide the reliability guarantees allowed by volumes because devices make no guarantee about what data resides on what devices, or how long they will keep the data they currently store. Views extend this notion by allowing volume-style reliability guarantees along with the flexibility of hoarding in the same abstraction.

A few filesystems suggested even more flexible methods of organizing data. BlueFS extended the hoarding primitive to allow client devices to access data hoarded on portable storage devices, in addition to the local device, but did not explore the use of this primitive for accessibility or reliability beyond that provided by Coda [21]. Footloose [24] proposed allowing individual devices to register for data types in this kind of system as an alternative to hoarding files, but did not expand it to general publish/subscribe-style queries, or explore how to use this primitive for mobility and reliability management or for distributed search.

Consistency: Perspective supports decentralized, topology-independent consistency for semantically-defined, partially replicated data, a critical feature for the home environment. While no previous system provides these properties out of the box, PRACTI [7] also provides a framework for topology-independent consistency of partially replicated data over directories, in addition to allowing a group of sophisticated consistency guarantees. PRACTI could probably be extended to use semantic groupings fairly simply, and thus provide consistency properties like Perspective. Recently, Cimbiosis [28] has also built on a view-style system of partial replication and topology independence, with a different consistency model.

Cimbiosis also presents a *sync tree* which provides a distributed algorithm to ensure connectedness, and routes updates in a more flexible manner. This sync tree could be layered on top of Perspective or PRACTI's consistency mechanisms to provide these advantages.

We chose our approach over Cimbiosis because it does not require any device to store all files, while Cimbiosis has this requirement. Many of the households in our contextual analysis did not have any such master device, leading us to believe requiring it could be a problem. Perspective also does not require small devices to track any information about the data stored on other devices, while PRACTI requires them to store imprecise summaries. However, there are advantages to each of these

approaches as well. For example, PRACTI provides a more flexible consistency model than Perspective, and Cimbiosis a more compact log structure. A full comparison of the differences between these approaches, and the relative importance of these differences, is beyond the scope of this paper. We present Perspective's algorithms to show that it is possible to build a simple, efficient consistency protocol for a view-based system.

Previous peer-to-peer systems such as Bayou [40], FICUS [16] and Pangaea [30] extended synchronization and consistency algorithms to accommodate mobile devices, allowing these systems to blur or eliminate the distinction between server and client. However, none of these systems fully support topology-independent consistency with partial replication. EnsemBlue [25] takes a middle ground, providing support for groups of client devices to form device ensembles [33], which can share data separately from a server through the creation of a temporary pseudo-server, but requiring a central server for consistency and reliability.

Search: We believe that effective home data management will use search on data attributes to allow flexible access to data across heterogeneous devices. Perspective takes the naming techniques of semantic systems and applies them to the replica management tasks of mobility and reliability as well. Naturally, Perspective borrows its semantic naming structures and search techniques from a rich history of previous work. The Semantic Filesystem [12] proposed the use of attribute queries to locate data in a file system, and subsequent systems showed how these techniques could be extended to include personalization [14]. Flamenco [43] uses "faceted metadata," a scheme much like the semantic filesystem's. Many newer systems [3, 13, 19, 37] borrow from the Semantic Filesystem by adding semantic information to filesystems with traditional hierarchical naming. Microsoft's proposed WinFS filesystem also incorporated semantic naming [42].

Perspective also uses views to provide efficient distributed search, by guiding searches to appropriate devices. The most similar work is HomeViews [11], which uses a primitive similar to Perspective's views to allow users to share read-only data. HomeViews combines capabilities with persistent queries to provide an extended version of search over data, but do not use them to target replica management tasks like reliability.

Replica indices and publish/subscribe: In order to provide replica coherence and remote data access, filesystems need a replica indexing system that forwards updates to the correct file replicas and locates the replicas of a given file when it is accessed remotely. Previous systems have used volumes to index replicas [32, 34], but

did not support replica indexing in a partially replicated peer-ensemble. EnsemBlue [25] extended the volume model to support partially replicated peer-ensembles by allowing devices to store a single copy of all replica locations onto a temporarily elected *pseudo-server* device. EnsemBlue also showed how its replica indexing system could be leveraged to provide more general application-level event notification. Perspective takes an inverse approach; it uses a publish/subscribe model to implement replica indexing and, thus, application-level event notification. This matches the semantic nature of views.

This work does not propose algorithms beyond the current publish/subscribe literature [1, 4, 6, 26, 38], it applies publish/subscribe algorithms to the new area of file system replica indices. Using a publish/subscribe method for replica indexing provides advantages over a pseudo-server scheme, such as efficient ensemble creation, but also disadvantages, such as requiring view changes to move replicas. Again, a full comparison of alternative approaches is beyond the scope of the paper. We present Perspective's algorithms to show that replica indexing can be performed efficiently using views.

User studies: While we believe our contextual analysis is the first focused on home data organization and reliability, researchers have conducted a wealth of studies on technology use and management, especially in the home [2, 5, 9, 15, 17, 20, 22, 39]. We borrow our methods from these previous studies, and use them to ground our exploration and analysis.

8 Conclusion

Home users struggle with replica management tasks that are normally handled by professional administrators in other environments. Perspective provides distributed storage for the home with a new approach to data location management: the view. Views simplify replica management tasks for home storage users, allowing them to use the same attribute-based naming style for such tasks as for their regular data navigation.

Acknowledgements

We thank Rob Reeder, Jay Melican, and Jay Hasbrouck for helping with the users studies. We also thank the members and companies of the PDL Consortium (including APC, Cisco, DataDomain, EMC, Facebook, Google, HP, Hitachi, IBM, Intel, LSI, Microsoft, NetApp, Oracle, Seagate, Sun, Symantec, and VMware) for their interest, insights, feedback, and support. This material is based on research sponsored in part by the National Science Foundation, via grants #CNS-0326453 and #CNS-

0831407, and by the Army Research Office, under agreement number DAAD19-02-1-0389. Brandon Salmon is supported in part by an Intel Fellowship.

References

- [1] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. *PODC*. (Atlanta, GA, 04–06 May. 1999), pages 53–61. ACM, 1999.
- [2] R. Aipperspach, T. Rattenbury, A. Woodruff, and J. Canny. A Quantitative Method for Revealing and Comparing Places in the Home. *UBICOMP* (Orange County, CA, Sep. 2006), 2006.
- [3] Beagle web page, <http://beagle-project.org>, 2007.
- [4] Sumeer Bhola, Yuanyuan Zhao, and Joshua Auerbach. Scalably supporting durable subscriptions in a publish/subscribe system. *DSN*. (San Francisco, CA, 22–25 Jun. 2003), pages 57–66. IEEE, 2003.
- [5] A. J. Bernheim Brush and Kori M. Inkpen. Yours, Mine and Ours? Sharing and Use of Technology in Domestic Environments. *UBICOMP 07*, 2007.
- [6] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service. *Nineteenth ACM Symposium on Principles of Distributed Computing (PODC2000)* (Portland, OR, Jul. 2000), pages 219–227, 2000.
- [7] Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramana, Praveen Yalagandula, and Jiandan Zheng. PRACTI Replication. *NSDI*. (May. 2006), 2006.
- [8] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. Persistent personal names for globally connected mobile devices. *OSDI*. (Seattle, WA, 06–08 Nov. 2006), pages 233–248. USENIX Association, 2006.
- [9] David M. Frohlich, Susan Dray, and Amy Silverman. Breaking up is hard to do: family perspectives on the future of the home PC. *International Journal of Human-Computer Studies*, **54**(5):701–724, May. 2001.
- [10] Filesystem in User Space. <http://fuse.sourceforge.net/>.
- [11] Roxana Geambasu, Magdalena Balazinska, Steven D. Gribble, and Henry M. Levy. HomeViews: Peer-to-Peer Middleware for Personal Data Sharing Applications. *SIGMOD*, 2007.
- [12] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole Jr. Semantic file systems. *SOSP*. (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review*, **25**(5):16–25, 13–16 Oct. 1991.
- [13] Google desktop web page, <http://desktop.google.com>, Aug. 2007.
- [14] Burra Gopal and Udi Manber. Integrating Content-based Access Mechanisms with Hierarchical File Systems. *OSDI*. (New Orleans, LA, Feb. 1999), 1999.
- [15] Rebecca E Grinter, W Keith Edwards, Mark W New-

- man, and Nicolas Ducheneaut. The work to make a home network work. *European Conference on Computer Supported Cooperative Work (ESCW)* (Paris, France, 18–22 Sep. 2005), 2005.
- [16] Richard G. Guy. *Ficus: A Very Large Scale Reliable Distributed File System*. PhD thesis, published as Ph.D. Thesis CSD-910018. University of California, Los Angeles, 1991.
- [17] Thomas Karagiannis, Elias Athanasopoulos, Christos Gkantsidis, and Peter Key. *HomeMaestro: Order from Chaos in Home Networks*. MSR-TR 2008-84. Microsoft Research, May. 2008.
- [18] Alexandros Karypidis and Spyros Lalas. OmniStore: A system for ubiquitous personal storage management. *IEEE International Conference on Pervasive Computing and Communications*. IEEE, 2006.
- [19] Dahlia Malkhi and Doug Terry. Concise Version Vectors in WinFS. *DISC*. (Cracow, Poland, Sep. 2005), 2005.
- [20] Catherine C Marshall. Rethinking Personal Digital Archiving, Part 1: Four Challenges from the Field. *DLib Magazine*, **14**(3/4), Mar. 2008.
- [21] Edmund B. Nightingale and Jason Flinn. Energy-efficiency and storage flexibility in the Blue file system. *OSDI*. (San Francisco, CA, 06–08 Dec. 2004), pages 363–378. USENIX Association, 2004.
- [22] Jon O’Brien, Tom Rodden, Mark Rouncefield, and John Hughes. At home with the technology: an ethnographic study of a set-top-box trial. *CHI*, 1999.
- [23] John K. Ousterhout, Andrew R. Cherenon, Fredrick Douglass, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, **21**(2):23–36, Feb. 1988.
- [24] Justin Mazzola Paluska, David Saff, Tom Yeh, and Kathryn Chen. Footloose: A Case for Physical Eventual Consistency and Selective Conflict Resolution. *IEEE Workshop on Mobile Computing Systems and Applications* (Monterey, CA, 09–10 Oct. 2003), 2003.
- [25] Daniel Peek and Jason Flinn. EnsemBlue: Integrating distributed storage and consumer electronics. *OSDI* (Seattle, WA, 06–08 Nov. 2006), 2006.
- [26] Peter R. Pietzuch and Jean M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. *International Workshop on Distributed Event-Based Systems* (Vienna, Austria), 2002.
- [27] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. *United Kingdom UNIX systems User Group* (London, UK, 9–13 Jul. 1990), pages 1–9. United Kingdom UNIX systems User Group, Buntingford, Herts, 1990.
- [28] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobblers, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: A Platform for content-based partial replication. *NSDI*. (Boston, MA, Apr. 2009), 2009.
- [29] Robert W. Reeder, Lujo Bauer, Lorrie Faith Cranor, Michael K. Reiter, Kelli Bacon, Keisha How, and Heather Strong. Expandable grids for visualizing and authoring computer security policies. *CHI* (Florence, Italy, 2007), 2007.
- [30] Yasushi Saito and Christos Karamanolis. *Name space consistency in the Pangaea wide-area file system*. HP Laboratories SSP Technical Report HPL-SSP-2002-12. HP Labs, Dec. 2002.
- [31] Brandon Salmon, Frank Hady, and Jay Melican. *Learning to Share: A Study of Sharing Among Home Storage Devices*. Technical Report CMU-PDL-07-107. Carnegie Mellon University, Oct. 2007.
- [32] M. Satyanarayanan. The evolution of Coda. *ACM Transactions on Computer Systems*, **20**(2):85–124. ACM Press, May. 2002.
- [33] Bill Schilit and Uttam Sengupta. Device Ensembles. *IEEE Computer*, **37**(12):56–64. IEEE, Dec. 2004.
- [34] Bob Sidebotham. VOLUMES – the Andrew file system data structuring primitive. *EUUGAutumn*. (Manchester, England, 22–24 Sep. 1986), pages 473–480. EUUG Secretariat, Owles Hall, Buntingford, Herts SG9 9PL, Sep. 1986.
- [35] Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Junwen Lai, Yilei Shao, Chi Zhang, Elisha Ziskind, Arvind Krishnamurthy, and Randolph Y. Wang. Segank: a distributed mobile storage system. *FAST*. (San Francisco, CA, 31 Mar.–02 Apr. 2004), pages 239–252. USENIX Association, 2004.
- [36] Craig A. N. Soules and Gregory R. Ganger. Connections: Using Context to Enhance File Search. *SOSP*. (Brighton, United Kingdom, 23–26 Oct. 2005), pages 119–132. ACM, 2005.
- [37] Spotlight web page, <http://www.apple.com/macosx/features/spotlight>, Aug. 2007.
- [38] Peter Sutton, Rhys Arkins, and Bill Segall. Supporting Disconnectedness - Transparent Information Delivery for Mobile and Invisible Computing. *International Symposium on Cluster Computing and the Grid (CCGrid)*, 2001.
- [39] Alex S. Taylor, Richard Harper, Laurel Swan, Shahram Izadi, Abigail Sellen, and Mark Perry. Homes that make us smart. *Personal and Ubiquitous Computing*. Springer London, 2006.
- [40] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *SOSP*. (Copper Mountain Resort, CO, 3–6 Dec. 1995). Published as *Operating Systems Review*, **29**(5), 1995.
- [41] Mark Weiser. The computer for the 21st century. *Scientific American*, Sep. 1991.
- [42] WinFS 101: Introducing the New Windows File System, March 2007. <http://msdn.microsoft.com/en-us/library/aa480687.aspx>.
- [43] Ping Yee, Kirsten Swearingen, Kevin Li, and Marti Hearst. Faceted Metadata for Image Search and Browsing. *CHI*, 2003.

BORG: Block-reORGanization for Self-optimizing Storage Systems

Medha Bhadkamkar^{*,§}, Jorge Guerra^{*,§}, Luis Useche^{*,§}, Sam Burnett[†], Jason Liptak[‡],
Raju Rangaswami[§], and Vagelis Hristidis[§]

[§]Florida International University, [†]Carnegie Mellon University, [‡]Syracuse University

Abstract

This paper presents the design, implementation, and evaluation of BORG, a self-optimizing storage system that performs *automatic block reorganization* based on the observed I/O workload. BORG is motivated by three characteristics of I/O workloads: non-uniform access frequency distribution, temporal locality, and partial determinism in non-sequential accesses. To achieve its objective, BORG manages a small, dedicated partition on the disk drive, with the goal of servicing a majority of the I/O requests from within this partition with significantly reduced seek and rotational delays. BORG is transparent to the rest of the storage stack, including applications, file system(s), and I/O schedulers, thereby requiring no or minimal modification to storage stack implementations. We evaluated a Linux implementation of BORG using several real-world workloads, including individual user desktop environments, a web-server, a virtual machine monitor, and an SVN server. These experiments comprehensively demonstrate BORG's effectiveness in improving I/O performance and its incurred resource overhead.

1 Introduction

There is a continual increase in the gap between CPU performance and disk drive performance. While the steady increase in main memory sizes attempts to bridge this gap, the impact is relatively small; Patterson *et al.* [25] have pointed out that disk drive capacities and workload working-set sizes tend to grow at a faster rate than memory sizes. Present day file systems, which control space allocation on the disk drive, employ static data layouts [5, 8, 15, 20, 22, 37]. Mostly, they aim to preserve the directory structure of the file system and optimize for sequential access to entire files. No file system today takes into account the dynamic characteristics of I/O workload within its data management mechanisms.

We conducted experiments to reconcile past observations about the nature of I/O workloads [7, 9, 30] in the context of current-day systems including end-user and server-class systems. Our key observations that motivate BORG are: (i) on-disk data exhibit a *non-uniform access frequency distribution*; the “frequently accessed” data is usually a small fraction of the total data stored when considering a coarse-granularity time-frame, (ii) considering

a fine-granularity time-frame, the “on-disk working-set” of typical I/O workloads is dynamic; nevertheless, workloads exhibit *temporal locality* in the data that they access, and (iii) I/O workloads exhibit *partial determinism* in their disk access patterns; besides sequential accesses to portions of files, fragments of the block access sequence that lead to non-sequential disk accesses also repeat. We elaborate on these observations in § 2.

While the above observations mostly validate the prior studies, and may even appear largely intuitive, surprisingly, there is a lack of commodity storage systems that utilize these observations to reduce I/O times. We believe that such systems do not exist because (i) key design and implementation issues related to the feasibility of such systems have not been resolved, and (ii) the scope of effectiveness of such systems has not been determined.

We built BORG, an online *Block-reORGanizing* storage system to comprehensively address the above issues. BORG correlates disk blocks based on block access patterns to capture the I/O workload characteristics. It manages a dedicated, *BORG OPTimized Target (BOPT)* partition and dynamically copies working-set data blocks (possibly spread over the entire disk) in their relative access sequence contiguously within this partition, thus simultaneously reducing seek and rotational delays. In addition, it assimilates all *write requests* into the BOPT partition's write buffer. Since BORG operates in the background it presents little interference to foreground applications. Also, BORG provides strong block-layer data consistency to upper layers, by maintaining a persistent page-level *indirection map*.

We evaluated a Linux implementation of BORG for a variety of workloads including a development workstation, an SVN server, a web server, a virtual machine monitor, as well as several individual desktop applications. The evaluation shows both the benefits and shortcomings of BORG as well as its resource overheads. Particularly, BORG can degrade performance when a non-sequential read workload suddenly shifts its on-disk working-set. For most workloads, however, BORG decreased disk busy times in the range 6% to 50%, offering the greatest benefit in the case of non-sequential write-mostly workloads without tuning BORG parameters for optimality. A sensitivity study with various parameters of BORG demonstrates the importance of careful pa-

*The first three authors contributed equally to this work.

Workload type	File System size [GB]	Memory size [GB]	Reads [GB]		Writes [GB]		File System accessed	Top 20% data access	Partial determinism
			Total	Unique	Total	Unique			
<i>office</i>	8.29	1.5	6.49	1.63	0.32	0.22	22.22 %	51.40 %	65.42 %
<i>developer</i>	45.59	2.0	3.82	2.57	10.46	3.96	14.32 %	60.27 %	61.56 %
<i>SVN server</i>	2.39	0.5	0.29	0.17	0.62	0.18	14.60 %	45.79 %	50.73 %
<i>web server</i>	169.54	0.5	21.07	7.32	2.24	0.33	4.51 %	59.50 %	15.55 %

Table 1: Summary statistics of week-long traces obtained from four different systems.

parameter choice which can lead to even greater improvements or degrade performance in the worst case; a self-configuring BORG is certainly a logical and feasible direction. Memory overheads of BORG are bound within 0.25% of BOPT, but CPU overheads are higher. Fortunately, most processing can be done in the background and there is ample room for improvement.

This paper makes the following contributions: (i) we study the characteristics of I/O workloads and show how the findings motivate BORG (§ 2), (ii) we motivate and present the detailed design and the first implementation of a disk data re-organizing system that adapts itself to changes in the I/O workload (§ 3 and § 4), (iii) we present the challenges faced in building such a system and our solutions to it (§ 5), and (iv) we evaluate the system to quantify its merits and weaknesses (§ 6).

2 Characteristics of I/O Workloads

In this section, we investigate the characteristics of modern I/O workloads, specifically elaborating on those that directly motivate BORG. We collected I/O traces, downstream of an active page cache, over a one-week period from four different machines. These machines have different I/O workloads, including *office* and *developer* desktop workloads, a version control *SVN* (*Subversion*) *server*, and a *web-server*. The office and developer workloads are single-user workloads. The former workload was composed mostly of web-browsing, graph plotting with gnuplot, and several open-office applications, while the latter consisted of extensive development using emacs, gcc, and gdb, document preparation using L^AT_EX, email, web-browsing, and updates of the operating system. The SVN server hosted document and project code-base repositories for our 6-person research group. Finally, the web-server workload mirrored the web-requests made to our department’s production web-server on one of our lab machines and served 1.1 million web requests during the trace period. Key statistics for these workloads are summarized in Table 1. We define the *on-disk working-set* (henceforth also referred to simply as “working-set”) of an I/O workload as the set of all unique blocks accessed in a given interval.

2.1 Non-uniform Access Frequency Distribution

Researchers have pointed out that file system data have non-uniform access frequency distribution [2, 29, 39].

This was confirmed in the traces that we collected where less than 4.5-22.3% of the file system data were accessed over the duration of an entire week (shown in Table 1). We observe that the office and web server workloads are read mostly, while the developer and SVN server are write mostly. Figure 1 (top row) shows page access rank-frequency plots for the workloads; file system pages were 4KB in size, composed of 8 contiguous blocks. A uniform trend to be observed across the various workloads is that the really high frequency accesses are due write requests. However, and especially in the case of the read-mostly office and web server workloads, there are a large number of read requests that occur repeatedly. In either case (read or write), the access frequencies are highly skewed. Figure 1 (middle row) depicts disk *heatmaps* created by partitioning the disk into regions and measuring accesses to each region. The heatmaps indicate that accesses, both high and low frequency ones, in most cases are spread over the entire disk area. Skewed data access frequency is further illustrated in Table 1 – the top 20% most frequently accessed blocks contributed to a substantially large (~45-66%) percentage of the total accesses across the workloads, which are within the ranges reported by Gómez and Santonja (Figure 2(a) in [7]) for the Cello traces they examined.

Based on the above observations, it is reasonable to expect that co-locating frequently accessed data in a small area of the disk would help reduce seek times when compared to the same data being spread throughout the entire disk area. Akyurek and Salem [2] have demonstrated the performance benefits of such an optimization via a simulation study. This observation also motivates reorganizing copies of popular blocks in BORG.

2.2 Temporal Locality

Temporal locality in I/O workloads is observed when the on-disk working-sets remain mostly static over short durations. Here, we refer to a locality of hours, days, or weeks, rather than seconds or minutes (typical of main memory accesses). For instance, a developer may work on a few projects over a period of a few weeks or months, typically resulting in her daily or weekly working sets being substantially smaller than her entire disk size. In servers, popularity of client requests result in temporal locality. A web server’s top-level links tend to be accessed more frequently than content that is embedded

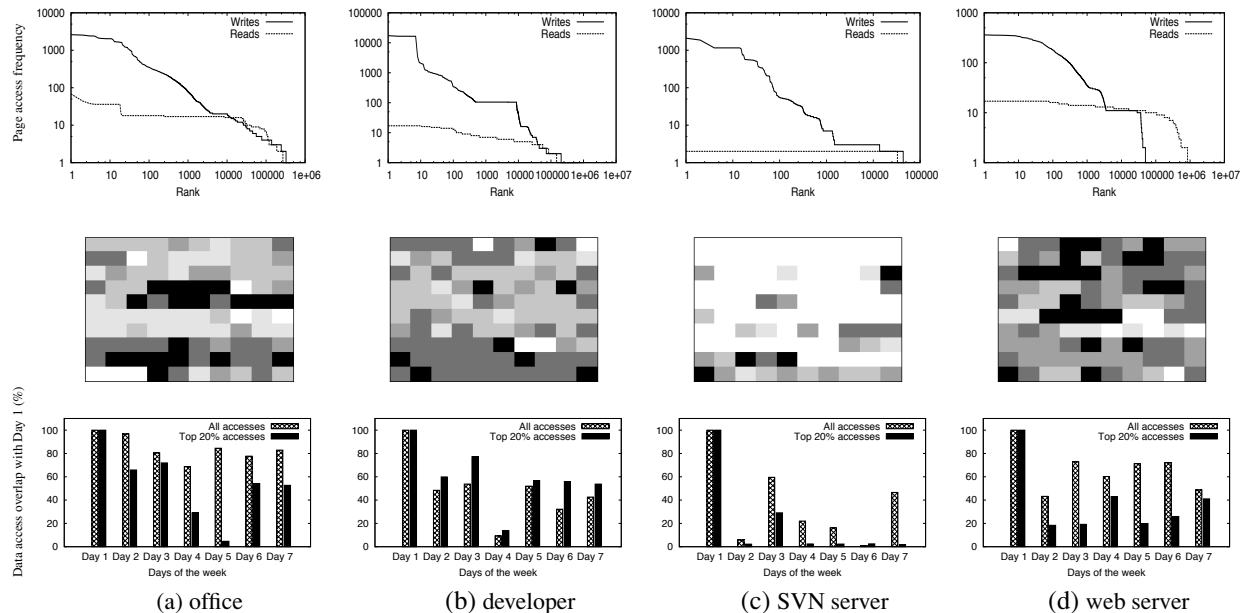


Figure 1: **Rank-frequency, heatmap, and working-set plots for week-long traces from four different systems.** The heatmaps (middle row) depict frequency of accesses in various physical regions of the disk, each cell representing a region. Six normalized, exponentially-increasing heat levels are used in each heatmap where darker cells represent higher frequency of accesses to the region. Disk regions are mapped to cells in row-major order.

much deeper in the web-site; an important new revision of a specific repository on an SVN server is likely to be accessed repeatedly over the initial weeks.

Figure 1 (bottom row) depicts the changes in the per-day working-sets of the I/O workload. The two end-user I/O workloads and the web server workload exhibit large overlaps in the data accessed across successive days of the week-long trace with the first day of the trace. There is substantial overlap even among the top 20% most accessed data across successive days. Interestingly, these workloads do not necessarily exhibit a gradual decay in working-set overlap with day 1 as one might expect, indicating that popularity is consistent across multi-day periods. The SVN server exhibits anomalous behavior because periods of high *commit* activity degrade temporal locality (new data gets created), while periods of high *update* activity improve temporal locality.

These observations indicate that optimizing layout based on past I/O activity can improve future I/O performance for some workloads and motivates planning block reorganization based on past activity in BORG.

2.3 Partial Determinism

Partial determinism in I/O workload occurs when certain non-sequential accesses in the block access sequence are found to repeat. A *non-sequential access* is defined by a sequence of two I/O operations that are addressed non-contiguous block addresses. It manifests in both end-user systems and servers. For instance, I/O during appli-

cation start-up is largely deterministic, both in terms of the set of I/O requests and the sequence in which they are requested. Reading files related to a repeatable task such as setting up a project in an integrated development environment, compilation, linking, word-processing, etc. result in a deterministic I/O pattern. In a web-server, accessing a web-page involves accessing associated sub-pages, images, scripts, etc., in deterministic order.

In Table 1, we present the partial determinism for each workload calculated as the percentage of non-sequential accesses that repeat at least once during the week. The partial determinism percentages are high for the two end-user and the SVN server workloads. Further, for each of these workloads, there were a non-trivial amount of non-sequential accesses that repeated as many as 100 times. These findings suggest that there is ample scope for optimizing the repeated non-sequential access patterns.

3 Overview and Architecture

BORG is motivated by the simple question: *What storage system optimizations based on workload characteristics can allow applications to utilize the disk drive more efficiently than current systems do?* This section presents the rationale behind the design decisions in BORG and its system architecture.

3.1 BORG Design Decisions

A Disk-based Cache.

The operating system uses main memory to cache fre-

quently and recently accessed file system data to reduce the number of disk accesses incurred. In any given duration of time, the effectiveness of the cache is largely dependent on the on-disk working-set of the I/O workload, and can degrade when this working-set increases beyond the size of the page cache. Storage optimizations such as prefetching [16, 24, 33] and I/O scheduling [13, 26, 27, 32] help improve disk I/O performance in such situations.

Using a disk-based cache as an extension of the main memory cache offers three complementary advantages in comparison to main memory caching alone, prefetching, and I/O scheduling. First, it is more effective as a cache (than main memory) because it offers a less expensive (and thus larger) as well as reliable caching solution, thus allowing data to be cache-resident for long periods of time. Second, the size of the disk-based cache can easily be configured by the system administrator without changing any hardware. And finally, dynamically optimizing data layout based on access patterns within a disk-based cache provides the unique ability to make originally non-sequential data accesses more sequential.

A Block Layer Solution.

A self-optimizing storage solution can be built at any layer in the storage stack (shown in Figure 2). Block level attributes of disk I/O operations are not easily obtained at the VFS or the page cache layer. While file system layer solutions can benefit from semantic knowledge of blocks, they incur a significant disadvantage in being tied to a specific file system (and perhaps even version). Device driver encapsulations (interface at P4) are incapable of capturing upper layer attributes, such as process ID and request time-stamp due to I/O scheduler re-ordering and loss of process context.

We contend that the block layer (interface at P3) is ideal for introducing block reorganization for several reasons. First, key temporal, block- and process- level attributes about disk accesses are available. Second, operating at the block layer makes the solution independent of the file system layer above, allowing it the flexibility to support multiple heterogeneous file systems simultaneously. Finally, new abstractions due to virtualization trends (e.g., virtual block device abstraction) as well as network-attached storage environments (SAN and NAS) can be supported in a straightforward way. In the case of SAN, BORG can reside on the client where all context for I/O operations are readily available with the underlying assumption that the SAN device's logical block address space is optimized for sequential access. In the case of NAS, the BORG layer can reside within the NAS device where I/O context is readily available. Modifying the NAS interface to include process associations within file I/O requests can complete the profile information.

Using an Independent BOPT partition.

The file system optimizes for sequential accesses to entire files, a common form of file access. However, certain workloads, including application start-up, content indexing and web-page requests, exhibit a more non-sequential, but deterministic, access behavior. It is thus possible that the same set of data can be accessed sequentially by some applications and non-sequentially by others. Further, some deterministic non-sequential accesses may only be temporary phenomenon.

Based on this observation, Akyurek and Salem [2] have argued in favor of *copying* rather than *shuffling* [29, 39] of data. Copying retains original sequential layouts so a choice of location based on the observed access pattern may be possible. Reverting back to the original layout is straightforward. Similarly, rather than permanently disturbing the sequential layout of files, BORG operates on copies of blocks placed temporarily in an independent BOPT partition, optimizing for the current common case of access for each data block.

3.2 BORG Architecture

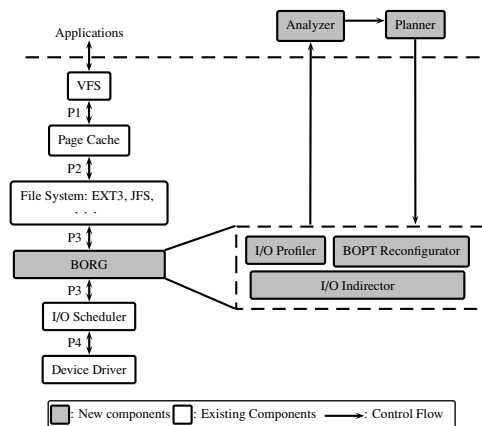


Figure 2: BORG System Architecture.

Abstractly, BORG follows a four-stage process:

1. *profiling* application block I/O accesses,
2. *analyzing* I/O accesses to derive access patterns,
3. *planning* a modification to the data layout, and
4. *executing* the plan to reconfigure the data layout.

In addition, an I/O indirection mechanism runs continuously, re-directing requests to the partition that it optimizes as required. Figure 2 presents the architecture of BORG in relation to the storage stack within the operating system. The modification to the existing storage stack is in the form of a new layer, which we term *BORG layer*, that implements three major components: the *I/O profiler*, the *BOPT reconfigurator* and the *I/O Indirector*. A secondary throttle-friendly user-space component implements the *analyzer* and the *planner* stages of BORG

and performs computation and memory-intensive tasks. While profiling and indirection are both continuous processes, the other stages run periodically and in succession culminating in a reconfiguration operation.

For the I/O profiler, we use a low-overhead kernel tool called `blktrace` [3]. The analyzer reads the I/O trace collected by the profiler and derives data access patterns. Subsequently, the planner uses these data access patterns and generates a new reconfiguration plan for the BOPT partition, which it communicates to the BOPT reconfigurator component. The user-space analyzer and planner components run as a low-priority process, utilizing only otherwise free system resources. Under heavy system load, the only impact to BORG is that generating the new reconfiguration plan would be delayed.

The BOPT reconfigurator is responsible for the periodic reconfiguration of the BOPT partition, per the *layout plan* specified by the planner. The reconfigurator issues low-priority disk I/Os to accomplish its task, minimizing the interference to foreground disk accesses. Finally, the I/O indirection continuously directs I/O requests either to the FS partition or the BOPT partition, based on the specifics of the request and the contents of the BOPT.

3.3 BOPT Space Management

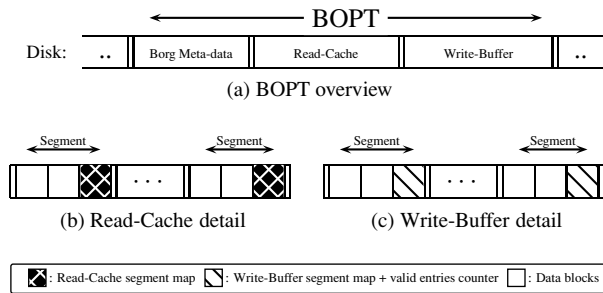


Figure 3: **Format of the BOPT partition.** Each entry in the Write-Buffer and Read-Cache map tables is a 3-tuple of the form (FS LBA, BOPT LBA, valid bit).

The Optimized Target partition (BOPT) as managed by BORG is shown in Figure 3. To reduce head movement, we suggest that the BOPT partition be created adjoining the *swap* partition if virtual memory is used. BORG partitions the BOPT into three fragments: *BORG Meta-data*, *Read-cache*, and *Write-buffer*. The Read-cache and Write-buffer are further sub-divided into fixed-length segments which store both data and (valid/invalid) map entries for the segment. The in-memory indirection map (elaborated in § 4.5) maintained by BORG is a union of all the segment map entries in the BOPT. The BOPT map entries are synchronously updated each time the in-memory map information changes. Additionally, the segment map in the write-buffer contains a “valid entries counter” to track space usage in the write buffer.

Magic number	BORG BOPTpartition identifier.
BORG_REQUIRE bit	BOPT contains dirty data.
BOPT size	BOPT partition size.
Read-cache info	Offset and size of the Read-cache.
Write-buffer info	Offset and size of the Write-buffer.
Segment size	Fixed size of segments in the BOPT.

Table 2: **Borg meta-data.**

Table 2 depicts the BOPT meta-data fragment. It stores key persistent information that aid in the operation of BORG. The `BORG_REQUIRE` bit is *set* when the BOPT contains data that requires to be copied back to the FS. If set, the operating system initiates BORG at boot time to ensure consistent data accesses. The remaining meta-data information is used to correctly populate the in-memory indirection map structure during BORG initialization.

4 Detailed Design

In this section, we present the design details of BORG by elaborating on its individual components.

4.1 I/O Profiler

The *I/O profiler* is a data collection component that is responsible for comprehensively capturing all disk I/O activity. The I/O profiler generates an *I/O trace* that includes the temporal (timestamp of the request), process (process ID and executable) and the block-level (address range and read/write mode) attributes. We use the `Q` events reported by `blktrace` [3], which capture the I/O requests queued at the block layer. These include all requests as issued by the file system(s), including any journaling and/or page destageing mechanisms. We defer further details to the `blktrace` work [3].

4.2 Analyzer

The *analyzer* is responsible for summarizing the disk I/O workload. It first splits the I/O trace obtained from the profiler into multiple I/O traces, one per process. Each process trace is used to build a directed *process access graph* $G_i(V_i, E_i)$, where vertices represent LBA ranges and edges a temporal dependency (correlation) between two LBA ranges. The weight on an edge between vertices (u, v) represents the frequency of accesses (reads or writes) from u to v . The *directed* and *weighted* graph representation is powerful enough to identify repeated sequences of multiple non-sequential requests.

Since multiple processes may access the same LBA, a single *master access graph* $G(V, E)$, that captures all available correlations into a single input for the reconfiguration planner is created (illustrated in Figure 4). The complexity of the merge process increases if two vertices (either within the same graph or across graphs) have overlapping ranges. This is resolved by creating multiple vertices so that each LBA is represented in at most one range vertex. While we omit the detailed algorithm

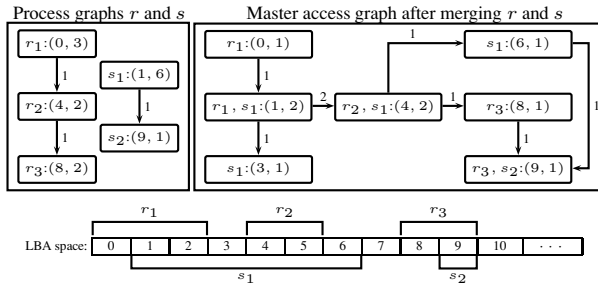


Figure 4: **Building the master access graph.** Vertices are defined by (start LBA, size of request). Since vertices r_1 and s_1 have overlapping LBAs, r_1 is split into two vertices in the master access graph, one with size 1 and the other with the overlapping s_1 blocks, starting at LBA 1 with size 2.

for vertex splitting and graph merging due to space constraints, we point out that we reduce the complexity of the merge algorithm by keeping the vertices sorted by their initial LBA. The total time complexity for the analyzer stage is given by $O(n \times l)$, where n is the number of vertices and l is the size (in LBA) of the largest vertex in the graph. Once the merge operation is completed, the master access graph, G , is obtained.

4.3 Planner

The *planner* takes the master access graph, G , as input and determines a reconfiguration plan for the BOPT partition. It uses a greedy heuristic that starts by choosing for placement the most connected vertex, u , i.e., with the maximum sum of incoming and outgoing edges (Figure 5). Next it chooses the vertex v most connected (in one direction only, either incoming or outgoing) to u . If v lies on the outgoing edge of u , it is placed after u and if it lies on the incoming edge it is placed before. The next vertex to be placed is the one most connected to the group $u \cup v$. This process is repeated until either all the vertices in G are placed, or the read cache in the BOPT is fully occupied, or the edges connecting to the unplaced vertices in the master graph have weight below a chosen threshold. If the graph contains disconnected components, each of these are placed as separate groups. The time complexity for the planner is $O(n \times \lg(m) + n^2)$ where n is the number of vertices and m is the number of edges; finding the most connected vertex takes $O(n \times \lg(m))$ time and finding the next vertex takes $O(n)$ time.

4.4 BOPT Reconfigurator

The *BOPT reconfigurator* implements the plan created by the planner component by performing the actual data movement to realize the new configuration of the BOPT. This task is complicated primarily because of consistency and overhead concerns. Overhead is partially addressed by issuing low-priority I/O requests for data lay-

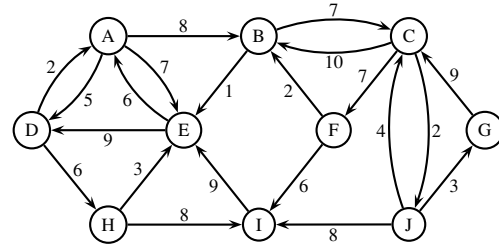


Figure 5: **Placing the master access graph.** C is the most connected vertex and is chosen for placement first. Next, vertex B is placed after vertex C since it is connected by an outgoing edge and has the highest weight of all the edges connected to C . Next, vertex G is placed before vertex group $C \cup B$. The final sequence of vertices from the lowest LBA to the highest is: $L = [F, H, J, A, G, C, B, E, D]$.

out reconfiguration, making the use of a priority scheduler a prerequisite. BORG ensures block data consistency between the FS and BOPT copies of data blocks by maintaining a persistent indirection map, termed the *borg_map*, that continuously tracks the most up-to-date location of a data block. This map is updated each time a block location changes.

The reconfigurator copies blocks in three stages; *outgoing*, where it copies all the dirty blocks that are no longer in the new plan back to the original file system (FS) location, *relocate*, where it copies blocks that have to be relocated within the BOPT, and *incoming* where it copies all the new blocks that have to be copied from the FS to the BOPT. A single data movement operation and the corresponding update on *borg_map* entry can be considered ‘atomic’ since any application *write* request to the *source* LBA during data movement is put on hold until after the movement is complete and the *borg_map* entry is updated. This ensures that an up-to-date version of data is always maintained by the file system.

4.5 I/O Indirector

The *I/O indirector* operates continuously, redirecting file system I/O requests as required. An I/O request may be composed of an arbitrary number of pages. Each page request is handled separately based on (i) number of blocks that can be satisfied from the BOPT as per the *borg_map* entry, (ii) type of operation (read or write) and (iii) presence of a free page in the BOPT.

For each I/O request larger than one page, the indirector splits it into multiple per-page requests. If a mapping exists for all the pages of the I/O request in the *borg_map*, the request is indirected to the BOPT. If no mapping exists, and the request is a read request, it is issued unchanged to the file system. If only some pages of a read I/O request are mapped and the mapped entries are clean, the entire I/O is indirected to the file system; this

optimization reduces the seek overhead incurred to serve the request partially from the BOPT and the rest from the FS. For a write request, when no mapping exists for any of the pages, the blocks are written to a *write-buffer* portion of the BOPT reserved for assimilating write requests (if space permits) along with an additional request for updating corresponding mapping entries in the `borg_map`. For partially-mapped writes, the mapped blocks are indirectioned to their BOPT locations; the unmapped pages are also absorbed in the write-buffer, space permitting, otherwise these are issued to the FS.

4.6 Kernel Data Structures

The persistent data structure `borg_map` is implemented as a radix tree such that given an FS LBA, the BOPT LBA can be retrieved efficiently and vice-versa. It also maintains the *dirty* information for the BOPT LBAs. For every page of 4KB, BORG stores 4 bytes each for the forward and the reverse mapping and one dirty bit. If all the pages in the BOPT of size S GB are occupied, the worst case memory requirement is $2 \times S$ MB (S MB for forward and reverse mapping each), and $\frac{S}{2^5}$ MB for the dirty information. Thus, in the worst case, `borg_map` requires memory of 0.25% of the size of the BOPT partition, an acceptable requirement for kernel-space memory.

5 Implementation Issues

In this section, we discuss the particularly challenging aspects of the BORG implementation that help address data consistency and overhead.

5.1 Persistent Indirection Map

Since BORG replicates popular data in the BOPT space, the system must ensure that reads are always up-to-date versions of data, including after a clean shutdown or a system crash. BORG implements a persistent `borg_map`, which is distributed within read-cache and write-buffer segments of the BOPT. Map entries on-disk are updated (along with their in-memory version) each time the BOPT partition is reconfigured or when a new map entry is added to accommodate a new write absorbed into the BOPT. Upon writes to an existing BOPT mapped block, its indirection entry in the in-memory copy of the reconfiguration map is marked as dirty, once the I/O is completed. To minimize overhead for BOPT writes, we chose not to maintain dirty information in the on-disk copy. Upon reboot after an unclean shut down, all entries in the persistent map are marked as dirty and future I/Os to these blocks are directed to the BOPT.

5.2 Optimizing Reconfiguration

Consider a set L of n LBAs, L_1, \dots, L_n , sequentially located in the BOPT space. L forms a *chain* if $\forall L_i \in L$, where $L_i \neq L_n$, L_i has to be relocated to location $L_i + 1$ and L_n is an outgoing block. If L_n has to be relocated to L_1 within the BOPT, L forms a *cycle*. Information about

chains and cycles, that occur exclusively for the relocated blocks, can be used to further optimize data movement during the reconfiguration operation. If a cycle exists, it is broken by copying the last block L_n back to the FS (if dirty) and then deleting the plan entry for that block; an additional plan entry is then created to mark this as incoming block to L_o . Next, all remaining blocks belonging to the same chain/cycle are copied to their new locations in the BOPT. To do so, the reconfigurator issues all reads to the source locations in parallel; once all reads have been completed, it issues all the writes in parallel, in each case allowing the I/O scheduler to optimize the request schedule.

5.3 Other Data Consistency Issues

BORG maintains metadata at the granularity of a *page* (rather than *block*) to reduce metadata memory requirement (by 8X for Linux file systems). Consequently, the indirector must carefully handle I/O requests whose sizes are not multiples of the page-size and/or which are not page-aligned to the beginning of the target partition. We address this issue via I/O request splitting and page-wise indirection, techniques borrowed from our earlier work on EXCES [38], a block-layer extension that manages a persistent cache for reducing disk power consumption.

BORG is dynamically included in the I/O stack by substituting the `make_request` function of the device targeted for performance optimization. While module insertion is straightforward, module removal/unload must ensure that all the data from the BOPT has been copied back to their original locations in the file system and handle foreground I/O correctly. Once again, BORG uses techniques from EXCES [38] and flushes dirty BOPT blocks to their original locations in the file system upon removal. To address race conditions caused when an application issues an I/O request to a page that is being flushed to disk, BORG stalls (via `sleep`) the foreground I/O operation until the specific page(s) being flushed are written to the disk.

6 Evaluation

In this section, we compare the performance of BORG with a *vanilla* system in which all the blocks are located in their original FS space under various workloads to answer the following questions.

(i) *How well does BORG perform?* We use the total disk busy time (i.e., excluding all idle periods) as the primary metric of performance. Due to BORG's optimizations, apart from the potentially improved head positioning times, the degree of merging of requests may also be increased when compared with the vanilla configuration, thus changing the request pattern itself. Thus, the more common I/O response time metric is an ill-suited choice. The total disk busy time (henceforth simply referred to as disk busy time) is also robust against the trace-replay

Host	Make	Model	RAM (MB)	Capacity (GB)		
				Total	FS	BOPT
O1	WD	2500AAKS	1024	250	46	1
O2	WD	360GD	1024	39	24	2
O3	Maxtor	6L020L1	1024	20	15	2
O4	WD	2500AAKS	1024	250	180	8
O5	Maxtor	6L020J1	1536	20	8	1

Table 3: Experimental test-bed details.

speedups we employ in some of our experiments.

(ii) *Why is BORG effective?* We would like to know if BORG performance gains are because of the sequentiality or the proximity of data (or both) in the BOPT. We use two metrics, *average seek distance* and *non-sequential accesses percentage* for this purpose. The latter is measured as $\frac{\# \text{ Seeks}}{\# \text{ BlocksRead}}$. Since non-sequential accesses are at least an order of magnitude less efficient than sequential accesses, even a small reduction in this metric may lead to substantial performance benefit.

(iii) *When is BORG not effective?* BORG can degrade the system performance for certain workloads. We evaluate BORG for varying workloads to determine in which cases it could perform worse than the vanilla system.

(iv) *How much CPU resource overhead does BORG incur?* While the upper bound on memory overhead was examined in § 4.6, the CPU resources consumed by BORG should also be within acceptable limits. We use the execution times for various stages of BORG as an approximate measure of CPU resource utilization.

(v) *How is BORG affected by its parameters?* We perform a sensitivity analysis of BORG to its parameters - reconfiguration interval, BOPT size, and BOPT write buffer fraction - to evaluate their impact on performance.

Experimental Setup. All experiments were performed on machines running the Linux 2.6.22 kernels. We used host machines, O1 through O5, with differing hardware configurations and disk drives (Table 3). We used `reiserfs` for O1 and O3, and `ext3` for the rest. No additional hardware was required to implement BORG.

We conducted four different sets of experiments. The first set uses week-long traces of a developer’s system and a Subversion control server (SVN). The second experiment is an actual deployment of a web server that mirrors our CS department’s web server. The third experiment evaluates BORG performance in a virtual machine environment. The fourth experiment evaluates the performance improvement due to BORG for application start-up events.

In each experiment, we performed 4 reconfigurations equally spaced in time; this gave us a reasonable number of phases for detailed analysis. By not choosing more favorable times such as idle disk periods based on well-known diurnal workload cycles, we would only overestimate the overhead of BORG during reconfiguration. We further discuss the selection of this parameter in § 6.5

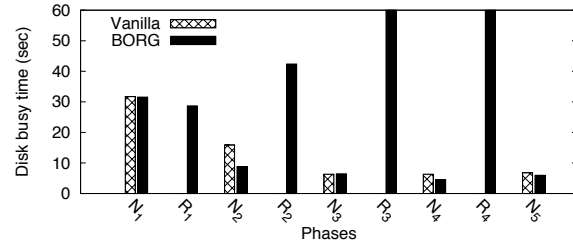


Figure 6: Disk busy times in various phases of the SVN server trace replay. R_i and N_j correspond to reconfiguration phase i and non-reconfiguration phase j respectively. R_3 and R_4 are beyond the y-axis range with values of 272 and 564 seconds respectively.

and § 7. Finally, we use the notation R_i and N_j in various graphs to denote reconfiguration phase i and non-reconfiguration phase j respectively.

6.1 Trace Replay

To evaluate BORG under realistic workloads, we conducted trace replay experiments using SVN server and developer workloads described in Table 1. For the traces and the replay, we used `blktrace` and `btoreplay` respectively [3]. We used an acceleration factor of 168X that reduces the experimentation time from one week to a manageable one hour after verifying that the resultant block access sequence was unaffected. The trace-playback acceleration factor was reverted to 1X during each reconfiguration operation to accurately estimate reconfiguration overhead. Since we only measure disk busy times, the comparison between normal and reconfigurations phases remains valid despite the varying acceleration factors.

6.1.1 SVN Server

For the SVN server trace replay, we used the host O2 (Table 3). The write buffer size was set to 20% of the BOPT size. Figure 6 shows the disk busy times during different phases of the experiment. In all the reconfiguration phases the busy time with BORG is notably higher than the vanilla case. This is due to substantial head movement during reconfiguration for relocating blocks. The longest reconfiguration phase lasted approximately 10 minutes. R_3 and R_4 have substantially higher busy time than the previous two reconfigurations. After trace analysis, we found that while the amount of data movement was similar across the four reconfiguration instances, in the latter two phases, the I/O scheduler merge ratio and the sequential disk accesses dropped dramatically; this can be attributed to the blocks relocated within the BOPT being spread out more than in the previous reconfigurations. However, As is evident by the vanilla busy times, the foreground activity during these intervals are negligi-

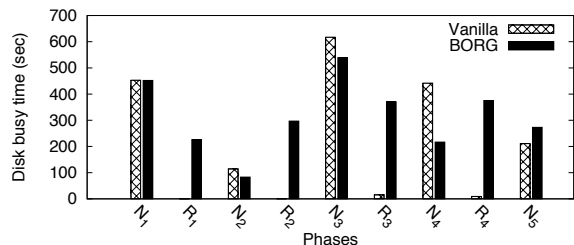


Figure 7: Disk busy time in various phases of the developer trace replay.

ble and thus the increased reconfiguration durations have little impact to foreground I/O.

In all the non-reconfiguration phases, each of which lasted 1.75 days approximately, BORG offers better performance for foreground I/O than the vanilla configuration. In the best case (range N_2), BORG decreases the disk busy time by approximately 45%. This is a surprising result, since as per Figure 1(c), the working-set for this workload undergoes rapid shifts. The explanation lies in the fact that the SVN server is a write-intensive workload and the BOPT write-buffer is successful in sequentializing a rapidly changing, possibly non-sequential, write workload. Analysis of the block level traces revealed that with BORG, the non-sequential access percentage reduced from 1.70% to 1.15%, and the average seek distance reduced from 704 to 201 cylinders during the non-reconfiguration phases.

6.1.2 Developer

For the developer trace replay, we used the host O1 (Table 3) with the BOPT write buffer set to 40% of the BOPT size. Figure 7 shows the disk busy time for this experiment in various phases. With this workload, the longest measured reconfiguration phases were R_3 and R_4 which lasted approximately 7 minutes each. We observe reduced disk busy times (13% to 50% reductions) across the non-reconfiguration periods, except for N_5 which shows an increase of 25%. Overall, the developer workload is a write-mostly workload and thus, largely conducive to BORG optimizations. Analysis of the block level traces revealed that overall, the non-sequential access percentage reduced from 3.93% to 3.30%, and the average seek distance reduced from 1203 to 782 cylinders when using BORG.

6.2 Web Server

To evaluate BORG in a production server environment, we made a copy of the our Computer Science department web server on the O4 machine (see Table 3), and replayed all the web requests for a week. During this week a total of 1137234 requests to 256017 distinct files were serviced. We set BORG to reconfigure four times

during this period, using an BOPT of 8GB (< 5% of the 180GB web server file system). To measure the influence of the I/O history, we conducted two sets of experiments. In the first experiment, we used all the traces gathered from the beginning of the experiment as input to the reconfigurator (*cumulative*). For the second, we only used the portion of the trace corresponding to the period since the last reconfiguration (*partial*).

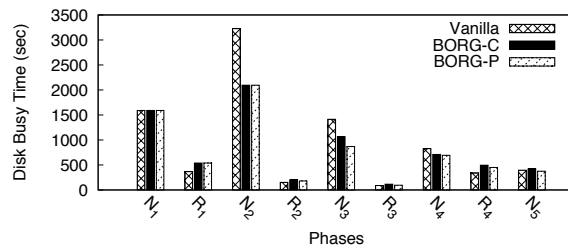


Figure 8: Disk busy time for the week long web log replay. Borg-C and Borg-P correspond to using cumulative and partial traces respectively.

Figure 8 shows the improvements in disk busy time across various non-reconfiguration and reconfiguration phases during the experiment. For both the cumulative and partial experiments, BORG reduces disk busy time in all non-reconfiguration phases with reductions ranging from 14% to 35% for cumulative and 5% to 39% for the partial configuration, except N_5 for cumulative which reported a 6% increase for cumulative due to drastic change in the last interval's workload. Disk busy times in reconfiguration phases are typically higher due to the overhead of copying data to the BOPT. Nevertheless, BORG was able to obtain overall reductions of 14% and 18% for cumulative and partial configuration. It is interesting to note that short term training yielded better results in this case, perhaps due to greater influence of short term locality.

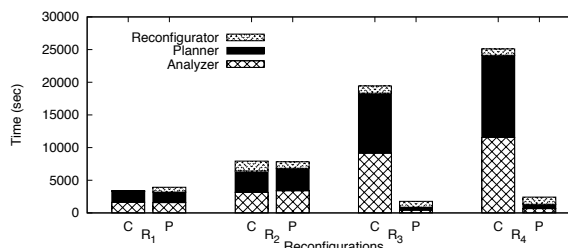


Figure 9: BORG overhead. Bars C and P represent the cumulative and partial traces experiments respectively. R_i indicates the i th reconfiguration.

Next we examine operational overhead of BORG. Figure 9 shows the amount of time spent in each phase of the reconfiguration. With cumulative traces, the time

required for the analyzer and planner phases increases linearly. While the planner and analyzer stages can run as low-priority tasks in the background, we must point out that the current implementation of BORG analyzer and planner stages are highly unoptimized and there is substantial room for improvement. We discuss possible improvements for both subsystems in §7. With partial traces, the time increases until the second reconfiguration, but then decreases and stays almost constant for the following ones, indicating a gradually stabilizing working-set.

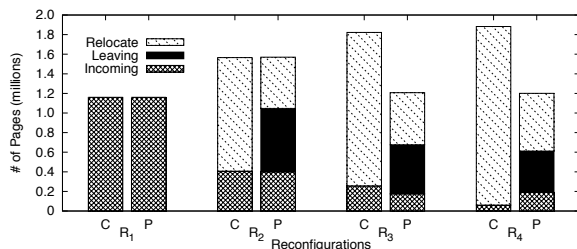


Figure 10: Differences in the reconfiguration plans.

To explain this further, we examined the reconfiguration plan divided by the type of operation (refer to § 4.4), presented in Figure 10. We note that the size of the plan consistently increases when using cumulative traces and most of the movements correspond to page relocates, which are page movements within the BOPT itself. The story is quite different for partial traces, where we see pages not accessed in the past interval leaving the BOPT, resulting in a smaller working set in the BOPT and thereby reducing the amount of work done by the analyzer, planner, and reconfigurator stages.

6.3 Virtual Machines

BORG has the potential to significantly improve the performance of virtualized environments, by co-locating multiple virtual machine (VM) localities spread across a physical volume. We evaluated the impact on the per-VM boot time and the overall performance of virtual machines by deploying BORG in a Xen [4] virtual machine monitor. We created four VMs, each with 64MB memory and 4GB physical partition on the host O5 (refer to Table 3). For evaluating boot-time improvement, we trained BORG with the boot-time events of all the virtual machines. BORG showed an almost 3X average improvement in VM boot-times - 167 seconds with vanilla and 65 seconds with BORG.

To measure normal execution performance improvement for the VMs, we ran the Postmark benchmark which emulates an e-mail server and creates and updates small files. We set the number of files to be 2000 in 500 directories and performed 200,000 transactions.

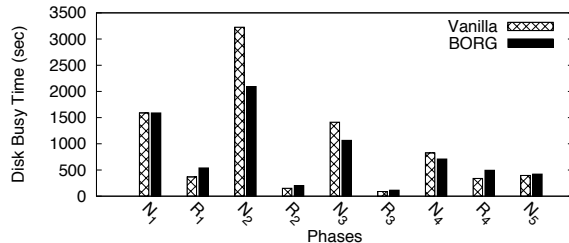


Figure 11: BORG with a VMM.

App	Start-up time		Rand. I/O %		Avg seek (#cyl)	
	V	B	V	B	V	B
firefox	3.71	2.32	2.7	1.2	132	37
oowriter	5.30	2.74	3.8	0.2	193	20
xemacs	7.26	2.72	2.1	0.3	87	9
acroread	6.20	2.65	4.6	0.1	39	9
eclipse	4.12	1.52	2.5	0.3	198	29
gimp	3.62	3.66	2.5	2.1	102	63
oaimpress	5.18	1.97	2.7	0.3	61	39

Table 4: Application start-up time improvement. V: vanilla, B: BORG.

We reconfigured BORG after every 20% of the benchmark was executed with the training set including I/O operations from the start of the execution of the benchmark. The results for the I/O performance are shown in Figure 11. As before, the reconfiguration phases see an increased disk busy times with BORG. For the normal operation, as the training set increases, the disk busy times with BORG starts decreasing. Overall, there is an average decrease of 6% in busy time during the non-reconfiguration phases. However, this improvement is not consistent; performance degrades substantially even during normal operation in the early stages of the benchmark. The *loss of process context* inside the VMM is a key problem that tends to convert sequentially laid out files into non-sequential upon reconfiguration. We believe that making BORG aware of process context inside the VMM [14] can substantially improve the BOPT layout, resulting in much greater performance benefit.

6.4 Application Start-up

We evaluated the impact of BORG on I/O-bound start-up phase for common desktop applications using host O3. We first trained the system for a duration of approximately four hours, during which we invoked a subset of the applications listed in Table 4 (but specifically excluding gimp and oaimpress) multiple times for performing common office tasks. We invalidated the page cache periodically to artificially dilate time and simulate system reboots. Table 4 shows the difference in application start-up times, the percentage of sequential accesses and average seek overhead. For the applications that were used in training, it can be observed that there is a noticeable improvement in the I/O time with BORG - at least 43% for oowriter and up to 67% for eclipse.

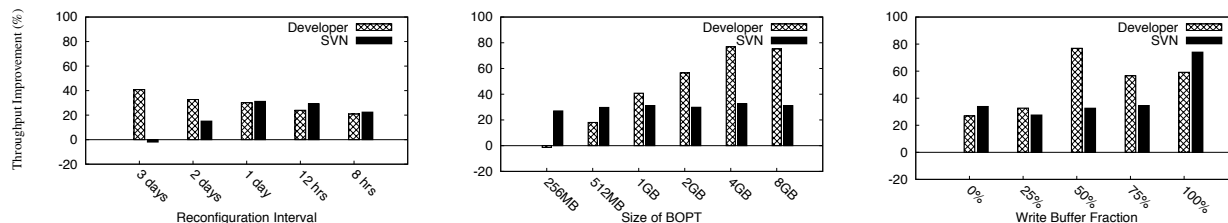


Figure 12: A sensitivity analysis of BORG performance to its configurable parameters.

Further, it is interesting to observe that although the percentage of sequential I/Os decreases for `oowriter` and `acoread` with BORG, there is an overall improvement in I/O performance, possibly due to a reduction in the rotational overhead. There is barely any difference in the performance for untrained application `gimp`. However, although `ooimpress` was not used in the training, its start-up user-time shows an improvement of 62% in the average I/O time; this can be attributed to large shared libraries also used by the `oowriter` which was included in training.

6.5 Sensitivity Analysis

To gain maximum performance improvement with BORG its configurable parameters – the reconfiguration interval, the BOPT size, and the BOPT write buffer fraction — must be carefully tuned for a given workload. To better understand the effects of these parameters, we replayed the developer and the SVN workload traces on host O1 varying each of these parameters over a range of values. In all the experiments, the trace replay begins at the same starting point, that is after a *base reconfiguration*, which uses the first six hours of the trace as the training period. We measure the relative efficiency of disk I/O using BORG averaged across the non-reconfiguration intervals by reporting the *improvement in disk busy time throughput* (referred to henceforth as “throughput improvement”) when compared to a vanilla system.

6.5.1 Reconfiguration Interval

Figure 12 (left) shows the percentage improvement over the vanilla system. The reconfiguration interval is varied from 8 hours (18 reconfigurations) to 3 days (1 reconfiguration). To bootstrap the sensitivity analysis, the BOPT size is fixed to 1GB, with 50% reserved for write buffering in this experiment. For the developer workload, as the reconfiguration interval increases the throughput increases, the training set becomes larger, and BORG can more effectively capture the working-set. For the SVN workload, the performance decreases for higher intervals. This is because the SVN working-set changes quite frequently (elaboration in § 2 and Figure 1(c)).

6.5.2 BOPT size

We use the best-case reconfiguration intervals of 3 days for the developer and a day for the SVN workload from the previous experiment. We vary the BOPT size from 256MB to 8GB, of which the write buffer is always chosen as 50% of the BOPT size. Figure 12 (middle) shows that as the BOPT size increases, BORG’s performance with the developer workload increases since the developer workload has a larger working set. When most of the blocks in the working set can be accommodated in the BOPT, the performance improvement stabilizes. Since the working set size for the SVN workload is relatively smaller, the performance improvement is almost same for the BOPT sizes >256MB.

6.5.3 Write Buffer Variation

From our previous results, we pick an interval of 3 days and 1 day and BOPT size of 2GB and 4GB for the developer and the SVN workloads respectively. We vary the write buffer from 0-100%. Figure 12 (right) shows that for the developer workload, not having a write buffer results in the lowest throughput. There is a steady increase in performance, peaking at 50% write buffer. Thereafter, it starts falling since read performance begins to degrade due to lesser available read cache. For the write-intensive SVN workload, the performance increases with increase in the write buffer size, since all the writes can be co-located in the BOPT partition.

Configuring BORG parameters The above experiments indicate that configuring parameters incorrectly can lead to sub-optimal performance improvements with BORG. Fortunately, iterative algorithms can be easily employed to identify better parameter combinations in a straightforward way. Exploring such iterative algorithms more formally is one aspect of our future work.

7 Discussion

While our experiences with BORG have been mostly positive, there are several directions in which the current version can be either improved or extended. We now discuss some of the significant directions that can serve as subjects of future investigation.

Analyzer and Planner optimization. The current versions of the analyzer (§ 4.2) and the planner (§ 4.3) com-

ponents of BORG do not use the results of past executions and therefore incur higher overheads for every subsequent reconfiguration when using cumulative traces for training. Each of these components can be substantially optimized by making them more intelligent. The analyzer can build the master access graph incrementally rather than from scratch; likewise, the planner can incrementally create the new plan for BOPT reconfiguration during each iteration.

Alternate BOPT layout strategies. The current version of BORG uses a simple BOPT layout strategy starting from the most-connected vertex – the vertex with the highest sum of its edge-weights – in the master access graph, and then choosing the vertex most connected to it, and so on. Alternate layout strategies can be envisioned that potentially yield greater benefit. For instance, the placement can begin with the nodes connected to the highest weight edge, and then resorting to the same incremental addition of vertices. Alternatively, a distributed layout algorithm can be designed which uses many starting points for building the layout.

Timely reconfiguration. The current reconfiguration trigger in BORG is based on a fixed interval. However, opportune times for performing reconfiguration are during periods of no or low foreground I/O activity, especially for workloads that exhibit obvious idle or peak periods of activity. More sophisticated triggers can use alternate metrics to identify “unwanted” or “much needed” reconfiguration, such as the BOPT hit rate or the percentage of sequential accesses pre- and post- indirection to evaluate the effectiveness of the current BOPT layout. The above techniques can help substantially reduce the impact of reconfiguration to foreground I/O and increase the effectiveness of each reconfiguration operation.

Avoiding performance degradation. BORG can degrade performance for certain workloads, for instance, a read-intensive workload that has a very large or unstable working-set (§ 6.2). Future versions of BORG can be made intelligent to measure the impact of reconfiguration on such workloads by comparing the percentage sequentiality and the spatial locality for the accesses before (vanilla) and after (BORG) the indirection operation. If these metrics degrade post-BORG, BORG can be disabled. Such a mechanism will allow system performance to degrade gracefully in the event that the workload is not conducive to benefit from block reorganization.

8 Related Work

We examine related work by organizing the literature into block and file based approaches.

8.1 Block level approaches

Early work [41] on optimized data layout argued for placing the frequently accessed data in the center of

the disk. Vongsathorn *et al.* [39] and Ruemmler and Wilkes [29] both propose Cylinder Shuffling. Ruemmler and Wilkes specifically demonstrated that performing relatively infrequent shuffling led to greater improvement in I/O performance. In Akyurek and Salem’s work [2], the authors demonstrated the advantages of copying over shuffling and the importance of reorganization at the block (rather than cylinder) level. These early data clustering approaches emphasized on process- and access-pattern- agnostic block counts to perform the data reorganization and reported simulation-based results.

Researchers have also investigate self-optimizing RAID systems. Wilkes *et al.* proposed HP AutoRAID [40], a controller-based solution, that transparently adapts to workload changes by using a two-level storage hierarchy; the upper level provides data redundancy for popular data while the lower level provides RAID 5 parity protection for inactive data. Work on eager writing [42] and distorted mirrors [35] address mirrored/striped RAID configurations primarily for database OLTP workload (which are characterized by little locality or sequentiality) that choose to write to a free sector closest to the head position on one more disk drives. While we are yet to explore BORG’s use in multi-disk systems, the optimizations used in BORG are different and mostly complementary to the above proposals, whereby BORG attempts to capture longer-term on-disk working-sets within a dedicated volume.

Hu *et al.*’s work on Disk Caching Disk [10] uses an additional logging disk (or disk partition) to perform writes sequentially and subsequently, destage to their original locations. Write buffering in BORG is slightly different in that writes to data already in the BOPT partition are written in place. The DCD work does not optimize for data read operations; BORG optimizes reads as well so head movement is substantially restricted.

Among recent work on block reorganization, C-Miner [17] uses advanced data mining techniques to mine correlations between block I/O requests. These techniques can be utilized in BORG to infer complex disk access patterns. The Intel Application Launch Accelerator [12] reorganizes blocks used during application start-up to be more sequential, but does not provide a generic solution to improve overall disk I/O performance of the system.

For throughput improvement, Schindler *et al.* have proposed free-block scheduling [18] and track-aligned extents [31] which use intelligent I/O scheduling rather than block reorganization. These are complementary techniques that can be used in conjunction with BORG.

Among block level approaches, our work is closest to ALIS [9], wherein frequently accessed blocks as well as block sequences are placed sequentially on a dedicated, reorganized area on the disk. There are key differences

in design and implementation, though. First, BORG incurs reduced space, maintenance, and metadata overhead since it maintains at most one copy of each data block. The multiple replicas in ALIS can become stale quickly in write-intensive workloads. Further, unlike BORG, ALIS does not optimize write traffic. Finally, the evaluation of ALIS techniques is performed using a disk simulator with trace playback. On the other hand, we implement and evaluate an actual system, thereby having the opportunity to address a greater detail of system implementation issues.

8.2 File level approaches

In one of the early file oriented approaches, Staelin *et al.* [36] proposed monitoring file accesses and moving frequently accessed files (entirely) to the center of the disk. Log-structured file systems (LFS [28]) offer superior performance for workloads with large number of small writes by batching disk writes to the end of a disk-sequential *log*. BORG writes all data to the BOPT partition to achieve a similar effect, but also attempts to co-locate a majority of read operations with the writes. Matthews *et al.* [19] proposed an optimization to LFS by incorporating data layout reorganization to improve read performance. Their use of block access graphs is similar to the process access graphs used in BORG. Their LFS-specific solution moves blocks within the LFS partition storing exactly one copy of each block at any time. Since BORG stores two copies, it can optimize for sequential and application-driven deterministic, non-sequential accesses simultaneously.

Researchers have also explored data- and application-specific layout mechanisms. Ganger and Kaashoek [6] advocate co-locating inodes and file blocks for small files. Conversely, PLACE [23], exposes the underlying layout structure to applications, so they can perform custom data placement. Sivathanu *et al.* [34] propose semantically-smart disk systems (SDS) that infer file system semantic associations for blocks, subsequently used for aligning files with track boundaries. Windows XP [21] uses the defragmenter for co-locating temporally correlated file data for speeding up application start-up events. BORG is a generic solution in comparison to the above approaches, since it creates a block reorganization mechanism that can adapt to an arbitrary workload.

Mac OS's HFS Plus [1] uses adaptive hot file clustering to migrate and sequentially store hot files of small sizes near the volume's metadata. In contrast, BORG operates at the block layer and sequentializes by copying (rather than migrating) hot block sequences, which may span either partial or multiple files.

Among file level approaches, BORG is closest to the FS2 [11]. FS2 proposes replication of frequently accessed blocks based on disk access patterns in file sys-

tem free space. This strategy, unfortunately, also restricts the degree of seek and rotational-delay optimization due to the distribution of free space. Since FS2 may create multiple copies of a block simultaneously, staleness, and consequently, space and I/O bandwidth wastage, become important concerns (similar to those in ALIS); BORG maintains at most one extra copy of each block and its strength is in being a non-intrusive, storage-stack friendly, and file system independent (portable) solution.

9 Conclusions and Future Work

We presented BORG, a self-optimizing layer in the storage stack that automatically reorganizes disk data layout to adapt to the workload's disk access patterns. BORG was designed to optimize both read and write traffic dynamically by making reads and writes more sequential and restricting majority of head movement within a small optimized disk partition. A Linux implementation of BORG was evaluated and shown to offer performance gains in the average case for varied workloads including office and developer class end-user systems, a web server, an SVN server, and a virtual machine monitor. Disk busy time reductions with BORG across these workloads during non-reconfiguration intervals range from 6% (for the VM workload) to 50% (for the developer server workload), with even greater improvements possible with careful parameter selection within BORG.

BORG performs occasionally worse than a vanilla system, specifically when a read-mostly workload drastically shifts its working set. BORG is able to easily address changing working-sets with a (possibly non-sequential) write workload, since it has the ability to absorb and sequentialize writes inside the BOPT. A sensitivity analysis revealed the importance of choosing the right configuration parameters for reconfiguration interval, BOPT size, and the write-buffer fraction. Fortunately, simple iterative algorithms can be quite effective in identifying the right parameter combination; a formal investigation of such an approach is an avenue for future work. The memory and CPU overheads incurred by BORG are modest, and with ample scope for further optimization. In summary, we believe that BORG offers a novel and practical approach to building self-optimizing storage systems that can offer large I/O performance improvements in commodity environments.

Acknowledgments

We would like to thank the reviewers of this paper and especially our shepherd Ken Salem for insightful feedback that helped improve the content and presentation of this paper substantially. This work was supported in part by the NSF grants CNS-0747038 and IIS-0534530 and by DoE grant DE-FG02-06ER25739.

References

- [1] HFS Plus Volume Format. <http://developer.apple.com/technotes/tn/tn1150.html>.
- [2] S. Akyurek and K. Salem. Adaptive Block Rearrangement. *Computer Systems*, 13(2):89–121, 1995.
- [3] J. Axboe. blktrace user guide, February 2007.
- [4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of the ACM SOSP*, October 2003.
- [5] H. Custer. Inside the Windows NT File System. *Microsoft Press*, August 1994.
- [6] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. *Proc. of the USENIX Technical Conference*, 1997.
- [7] M. Gómez and V. Santonja. Characterizing Temporal Locality in I/O Workload. *Proc. of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2002.
- [8] M. Holton and R. Das. XFS: A Next Generation Journalled 64-bit filesystem with Guaranteed Rate IO. *SGI Technical Report*, 1996.
- [9] W. W. Hsu, A. J. Smith, and H. C. Young. The automatic improvement of locality in storage systems. *ACM Transactions on Computer Systems*, 23(4):424–473, Nov 2005.
- [10] Y. Hu and Q. Yang. DCD – Disk Caching Disk: A New Approach for Boosting I/O Performance. *Proc. of the International Symposium on Computer Architecture*, 1995.
- [11] H. Huang, W. Hung, and K. G. Shin. FS2: Dynamic Data Replication In Free Disk Space For Improving Disk Performance And Energy Consumption. *Proc. of the ACM SOSP*, October 2005.
- [12] Intel Corporation. Intel application launch accelerator. <http://support.intel.com/support/chipsets/iaa/>, 1998.
- [13] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. *Proc. of the ACM SOSP*, Sept 2001.
- [14] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. *Proc. of the USENIX Technical Conference*, May 2006.
- [15] D. Kleikam, D. Blaschke, S. Best, and B. Arndt. JFS for Linux. <http://jfs.sourceforge.net/>.
- [16] C. Li and K. Shen. Managing Prefetch Memory for Data-Intensive Online Servers. *Proc. of the USENIX FAST*, December 2005.
- [17] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou. C-Miner: Mining Block Correlations in Storage Systems. *Proc. of the USENIX FAST*, April 2004.
- [18] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock Scheduling Outside of Disk Firmware. *Proc. of USENIX FAST*, January 2002.
- [19] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. *Proc. of the ACM SOSP*, 1997.
- [20] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX*. *ACM Transactions on Computer Systems* 2, 3:181–197, August 1984.
- [21] Microsoft Corporation. Fast System Startup for PCs Running Windows XP. *Windows Platform Design Notes*, December 2006.
- [22] Namesys, Inc. The ReiserFS File System. <http://www.namesys.com/>.
- [23] J. Nugent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Controlling your PLACE in the File System with Gray-box Techniques. *Proc. of the USENIX Technical Conference*, June 2003.
- [24] A. E. Papathanasiou and M. L. Scott. Aggressive Prefetching: An Idea Whose Time Has Come. *Proc. of the Workshop on HotOS*, June 2005.
- [25] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proc. of the 15th ACM SOSP*, December 1995.
- [26] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. *Proc. of the USENIX Technical Conference*, June 2003.
- [27] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). *Proc. of the IEEE RTSS*, December 2003.
- [28] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *Proc. of the ACM SOSP*, October 1991.
- [29] C. Ruemmler and J. Wilkes. Disk Shuffling. *Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories*, October 1991.
- [30] C. Ruemmler and J. Wilkes. UNIX disk access patterns. *Proc. of the Winter USENIX Conference*, 1993.
- [31] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. *Proc. of USENIX FAST*, 2002.
- [32] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisited. *Proc. of the Winter USENIX Technical Conference*, 1990.
- [33] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Operating Systems. *Proc. of the Workshop on HotOS*, May 1997.
- [34] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. *Proc. of the USENIX FAST*, March 2003.
- [35] J. A. Solworth and C. U. Orji. Distorted Mirrors. *Proc. of PDIS*, 1991.
- [36] C. Staelin and H. Garcia-Molina. Smart Filesystems. In *USENIX Winter Conference*, 1991.
- [37] S. C. Tweedie. Journaling the Linux ext2fs File System. *The Fourth Annual Linux Expo*, May 1998.
- [38] L. Useche, J. Guerra, M. Bhadkamkar, M. Alarcon, and R. Rangaswami. EXCES: External caching in energy saving storage systems. *IEEE HPCA*, 2008.
- [39] P. Vongsathorn and S. D. Carson. A System for Adaptive Disk Rearrangement. *Softw. Pract. Exper.*, 20(3):225–242, 1990.
- [40] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *Proc. of the ACM SOSP*, 1995.
- [41] C. K. Wong. Minimizing Expected Head Movement in One-Dimensional and Two-Dimensional Mass Storage Systems. *ACM Computing Surveys*, 12(2):167–178, 1980.
- [42] C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang. Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload. *Proc. of USENIX FAST*, January 2002.

HYDRAsTOR: a Scalable Secondary Storage

Cezary Dubnicki[§], Leszek Gryz[§], Lukasz Heldt[§], Michal Kaczmarczyk[§], Wojciech Kilian[§],
Przemyslaw Strzelczak[§], Jerzy Szczepkowski[§], Cristian Ungureanu, and Michal Welnicki[§],

NEC Laboratories America

[§]formerly at NEC Laboratories America, now at 9LivesData, LLC

{dubnicki, gryz, heldt, kaczmarczyk, wkilian, strzelczak, jsz, welnicki}@9livesdata.com, cristian@nec-labs.com

Abstract

HYDRAsTOR is a scalable, secondary storage solution aimed at the enterprise market. The system consists of a back-end architected as a grid of storage nodes built around a distributed hash table; and a front-end consisting of a layer of access nodes which implement a traditional file system interface and can be scaled in number for increased performance.

This paper concentrates on the back-end which is, to our knowledge, the first commercial implementation of a scalable, high-performance content-addressable secondary storage delivering global duplicate elimination, per-block user-selectable failure resiliency, self-maintenance including automatic recovery from failures with data and network overlay rebuilding.

The back-end programming model is based on an abstraction of a sea of variable-sized, content-addressed, immutable, highly-resilient data blocks organized in a DAG (directed acyclic graph). This model is exported with a low-level API allowing clients to implement new access protocols and to add them to the system on-line. The API has been validated with an implementation of the file system interface.

The critical factor for meeting the design targets has been the selection of proper data organization based on redundant chains of data containers. We present this organization in detail and describe how it is used to deliver required data services. Surprisingly, the most complex to deliver turned out to be on-demand data deletion, followed (not surprisingly) by the management of data consistency and integrity.

1 Introduction

The enterprise environment places strenuous demands on the secondary storage systems. With ever increasing amounts of data produced and fixed backup windows, there is a clear need for scaling performance and

backup capacity appropriately. Different types of data have varying importance which require different classes of reliability and availability and have specific retention periods. Regulatory requirements (SOX, HIPPA, the Patriot Act, SEC rule 17a-4(t)) demand security, traceability and data auditing. Strict data retention and deletion procedures need to be defined and followed rigorously. Failure to present retained data on demand can result in serious business losses, fines and even criminal prosecution. Last but not least, limited IT budgets increase the importance of providing efficient storage by improving storage utilization for backup and archival applications, and by reducing the data management costs.

Substantial progress has been made to address these enterprise needs, as demonstrated by advanced disk-targeted deduplicating Virtual Tape Libraries [2, 3], disk-based back-end servers [43] and content-addressable archiving solutions [4]. However, the exponential increase in the amount of data stored creates new problems not addressed by these solutions. First of all, unlike primary storage, which is usually networked and under common management (e.g. SANs), secondary storage consists of a large number of highly-specialized dedicated components, each of them being a storage island requiring customized, elaborate, and often manual administration and management. As a result, large fraction of the total cost of ownership (TCO) can still be attributed to management of more and more of secondary storage components [1, 17, 21]. Moreover, fixed capacity assignment to each storage device results in poor capacity utilization. Duplicate elimination in these islands of storage is similarly limited in scope which compounds the inefficiency. Finally, since each of secondary storage devices offers fixed, limited performance, reliability and availability, the high overall requirements of enterprise secondary storage in these dimensions can be met only by implementing complex in-house solutions.

Fortunately, new technology and previous research results provide building blocks for a solution address-

ing these problems. The content-addressable storage paradigm [4, 24, 43] enables cheap and safe implementation of duplicate elimination. Distributed hash tables [12, 20, 25, 27, 31, 42] allow for building scalable, failure-resistant systems and extending duplicate elimination to a global level. Erasure codes can add resiliency to the stored data with fine-grain control between required resiliency level and resulting storage overhead. Hardware and pricing trends are also critical for enabling HYDRAsstor. The capacity of SATA drives and the performance of new multi-core CPUs increase even as the costs of these components fall. Together, these trends provide the building blocks needed for systems like HYDRAsstor at a very reasonable cost.

Other work applicable include research on self-management [13, 33], monitoring [35], and on-line re-configuration [30] and upgrade [7]. Although all of these elements facilitated building HYDRAsstor, the task proved to be much more complex than we originally envisioned and required a significant amount of original research. The effort often felt like trying to design and construct a building given just bricks and stones.

HYDRAsstor [23] is a commercial secondary storage solution for the enterprise addressing shortcomings discussed earlier. It consists of a back-end architected as a grid of storage nodes delivering scalable capacity and a front-end consisting of a layer of access nodes scaled for performance. In this paper, we concentrate on the design of the back-end grid which supports capacity sharing between all clients and types of data, for example, back up images or archival data. This sharing together with system-wide deduplication allow for highly efficient use of storage capacity. The system is highly-available, as it supports on-line extensions and upgrades, tolerates multiple disk, node and network failures, rebuilds the data automatically after failures and informs users about recoverability of the deposited data. The reliability and availability of the stored data can be dynamically adjusted by the clients with each write, as the back-end supports multiple data resiliency classes.

This paper makes the following contributions. First, it presents the HYDRAsstor as a concrete commercial implementation of scalable secondary storage system addressing today's enterprise needs. Second, it discusses in detail the HYDRAsstor data organization and how it is used to implement advanced data services like global duplicate elimination, on-demand deletion, and data integrity management. Third, it contains an evaluation of the HYDRAsstor that demonstrates effectiveness of its implementation.

The remainder of this paper is organized as follows. Section 2 describes the system's functionality including the programming interface. Section 3 contains a high-level discussion of the back-end design. It establishes

context for the next section, 4, which discusses requirements on data organization and the resulting solution. Section 5 illustrates how this organization is used to deliver data services like data rebuilding and distributed data deletion. Section 6 presents evaluation of the system. Related work is discussed in Section 7, whereas conclusions and future work are given in Section 8.

2 Functionality

The back-end has been designed as a vast data repository, allowing for storing and extracting streams of data with high throughput. Internally, it consists of a potentially large number of independent nodes presented externally as a single system image. The back-end is designed to scale up to thousands of dedicated nodes which could provide hundreds of petabytes of storage. The primary deployment target is the data center.

From the beginning, the HYDRAsstor back-end was intended to provide a foundation for a commercial product. Therefore, one of the design targets has been to support not only tailor-made new applications, but also commercial legacy applications, as long as they use streamed data access. To that end, the system does not define one fixed access protocol, instead it is flexible to allow support for legacy applications using standards like file system interface as well as for new applications using highly-specialized access methods. New protocols can be dynamically added to an online system by loading a new protocol driver without disrupting any client using the existing protocols.

One of the primary design goals has been to ensure continuous operation of the system, limiting or eliminating impact of upgrades, extensions and failures. The distributed architecture enhances system availability by allowing online software or hardware upgrades in most cases, eliminating the need for costly downtime. Moreover, the system is capable of automatic self-recovery in case of hardware failures (disk, network, power loss), and even from some of software failures. The system works correctly in the presence of up to a specific configurable number of fail-stop and intermittent hardware failures. The system does not handle Byzantine failures which have a very low probability of actually occurring in a real data center and would add significant overhead. However, the system has several layers of data integrity checking to detect data corruption.

Another important function of the system is to ensure high data reliability, availability and integrity. Each block of data is written with a user-selected resiliency level which allows the user to choose how many concurrent disk failures the block can survive. This is achieved with erasure coding each block into fragments; as shown in [36] erasure codes increase mean time to failure by

many orders of magnitude over simple replication for the same amount of space overhead. After a failure, if a block remains readable, the system automatically schedules data rebuilding to bring the resiliency back to the level requested by the user. No permanent data loss remains hidden for long. Global state indicates whether all stored blocks are readable, and if so, how many disk and node failures must happen before data loss occurs.

Secondary storage systems have unique characteristics which influence the design goals. In contrast to primary storage, which often deals with random accesses, these systems are dominated by writes of long data streams. Given the scale of the system, multiple streams will be written concurrently by different clients. Successive streams are often similar to previously written streams which can contain many duplicate blocks. Since all data must be saved during short backup windows, very high write throughput is essential. Read throughput is quite important for restores, but it is not as critical as write throughput in our system since the restores are typically much less frequent and involve reading only a portion of the stored data.

2.1 Programming Model

The back-end programming model is based on an abstraction of a sea of variable-sized, content-addressed, immutable, highly-resilient blocks. A block consists of data and, optionally, an array of block addresses, pointing to previously written blocks. A block's address is derived from the SHA-1 hash of its content (both data and pointers). Blocks are variable-sized to allow for better deduplication; and pointers are exposed to facilitate data deletion implemented as garbage collection. The back-end exports a low-level block interface used to implement new and legacy protocols. This interface allows for a clean separation of the back-end from the front-end which can support a wide range of access protocols.

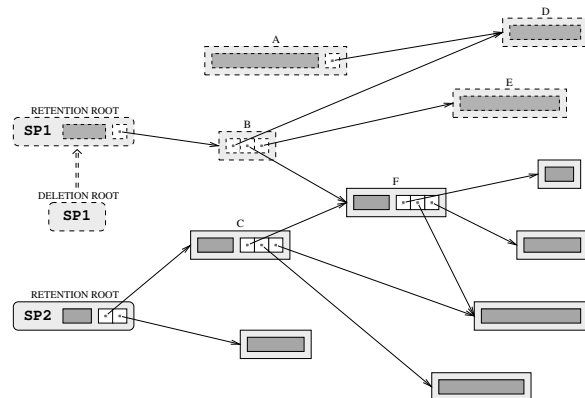


Figure 1: Blocks organized in a directed acyclic graph. Data part of each block is shaded, pointers are not.

Blocks in the back-end form a DAG (directed acyclic graph), as illustrated by Fig. 1. Drivers write trees of blocks, but because of deduplication, these trees overlap at deduplicated blocks and form directed graphs. No cycle is possible in these structures as long as the hash used in the block address is secure. A source vertex in a DAG is usually a block of a special type called *searchable retention root*. In addition to the regular data and the array of addresses, a retention root contains a user-defined *search key* used to locate the block. This key can be arbitrary data. A user retrieves a searchable block by providing its search key instead of a cryptic block content address. For example, multiple snapshots of the same file system can have each root organized as a searchable retention root with search key containing file system name and a counter incremented with each snapshot. Searchable blocks do not have user-visible addresses and cannot be pointed to, so they cannot be used to create cycles in block structures. However, each searchable block has an internal hashkey assigned to it for fast retrieval. Unlike regular blocks, the hashkey of a searchable block is computed only over the search key portion of the block's data.

Fig. 1 shows a set of blocks organized into a DAG with 3 source vertices, 2 of them are retention roots; the 3rd source vertex is a regular block, which indicates that this part of the DAG is still under construction.

The API operations include writing and reading regular blocks, writing searchable retention roots, searching for a retention root based on its search key; and marking a retention root with a specified key to be deleted by writing an associated deletion root, as discussed below. Cutting the data stream into blocks is beyond this interface and is responsibility of the drivers, although we plan to re-evaluate this decision soon.

When writing a block, a driver assigns it to one of a few available *resiliency classes*. Each class represents a different tradeoff between data resiliency and storage overhead: from the low resiliency data class where a block can survive only a single disk failure but has minimum storage overhead, up to the critical data class where each block can be replicated multiple times on different disks and physical nodes. Different resiliency classes are achieved by varying the number of original fragments in the erasure coding scheme (described later).

The system does not provide a way to delete a single block immediately because this block may be referenced by other blocks. Instead, the API allows to mark roots of the DAG(s) which should be deleted. To mark a retention root as dead, a user writes a special block called *searchable deletion root* with the search key identical to this retention root's search key. In Fig. 1, there is a deletion root associated with the retention root SP1. The deletion algorithm marks for deletion all blocks not

reachable from the live retention roots, for example in Fig. 1 all blocks with dotted lines will be marked. The block named A will also be deleted because there is no retention root pointing to it, whereas the block named F will be retained, as it is reachable from the retention root SP2 which is still defined as live since it does not have a matching deletion root.

During data deletion, there is a short read-only period, in which the system identifies blocks to be deleted. Actual space reclamation happens in the background during regular read-write operation. Before entering a read-only phase, all blocks to be retained should be pointed by live retention roots.

3 System Architecture

HYDRAsstor back-end nodes are built of highly reliable server-grade components. No customized hardware is needed. Detailed description of available hardware configurations is given in Section 6. The number of storage nodes determines the total raw capacity of the system as well as its maximal level of performance. Front-end access nodes can be added to realize this performance up to the limit determined by the current back-end configuration.

Software components of the back-end include the *storage server* and *proxy server*, both implemented as Linux user space processes, and *protocol drivers* implemented as libraries.

Storage servers are organized in an overlay network, with data blocks assigned to each server based on block's hashkey. The details of the overlay are discussed in Section 3.1. Each storage node hosts one or more storage servers. The number of storage servers running on a storage node depends on its resources. The bigger the node, the more servers we run, with each server responsible exclusively for a specific number of this node's disks. Putting multiple servers on one physical node is a simple solution to the problem of harnessing computing power of multicore CPUs.

Proxy servers run on access nodes and export the same block API as the storage servers. A proxy provides services like locating the storage nodes, optimized message routing and caching.

Protocol drivers use the API exported by the back-end to implement access protocols. These drivers can be loaded in the runtime on both storage and proxy servers. Location of a driver depends on available resources and driver resource needs. Usually, resource-hungry drivers like the file system driver are loaded on proxy servers.

3.1 Network Overlay

Since one of our design goals has been scalability, the use of distributed hash tables has been a natural choice. However, because for a distributed storage system both storage utilization and data resiliency are extremely important, we have had additional requirements on a DHT: assurances about storage utilization and ease of integration of the selected overlay network with the data resiliency scheme we have planned to use, i.e. erasure coding. Since none of the existing DHTs allowed for that, we have decided to use a modified version of the *Fixed Prefix Network* (FPN) [12] distributed hash table. FPN makes it possible to maintain very short routing paths for a wide range of the number of nodes and guarantees a minimal level of storage utilization.

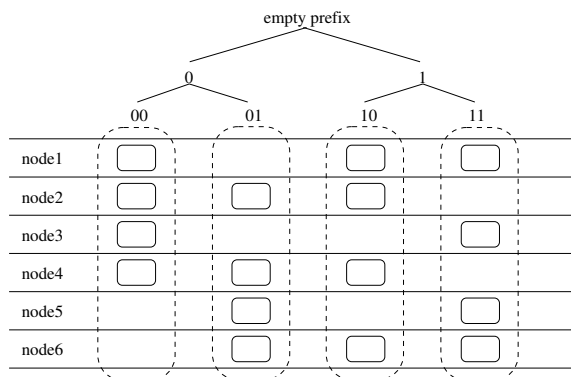


Figure 2: Supernodes and components. 4 supernodes spanned over 6 physical nodes. Each supernode has 4 components, i.e. supernode cardinality is 4.

In FPN, each overlay node is assigned exactly one hashkey prefix used also as an identifier of this virtual node. An FPN node is responsible for hashkeys with prefix equal to this node identifier. All possible hashkeys form a hashkey space. The overlay network strives to keep prefixes disjoint and to cover completely this space, which we call also *prefix space*. The upper part of Fig. 2 shows a prefix tree which has four leaf FPN nodes, dividing the prefix space into four disjoint subspaces.

To meet our DHT requirements, we have extended the original FPN with *supernodes*. A supernode represents one FPN node (and as such, it is identified with a hashkey prefix), but spans several physical nodes to increase resiliency to node failures. Each supernode consists of a fixed number (called *supernode cardinality*) of *supernode components*. Components of the same supernode are called *peers* and are usually placed on separate physical nodes, as show on Fig. 2. Practical supernode cardinality values are in the 4-32 range, and in the commercial HYDRAsstor it is set to 12. For a given HYDRAsstor incarnation, its supernode cardinality is the same for all

supernodes and is constant throughout entire system lifetime.

Supernode peers use a distributed consensus algorithm to decide what change should be applied to the supernode — for example, after node failure, they decide on which physical nodes new incarnations of lost components should be recovered.

3.2 Read and Write Handling

On write, a block of data is routed to one of the peers of the supernode responsible for the hashkey space where the block's hash belongs. For both read and write requests, the peer is deterministically chosen based on the hashkey of the data. Next, this write-handling peer checks if a suitable duplicate is already stored; this process is described in detail in Section 5.2. If a duplicate is found, its address is returned; otherwise the new block is compressed (if requested by a user), fragmented, and its fragments are distributed to remaining peers.

On read, the read handling peer first determines the minimal number of fragments required to reconstruct this block, which is stored in the block's metadata. Next, the read handling peer sends fragment read requests to some of the other peers. If any of these requests times out, all remaining fragments are read. After a sufficient number of fragments have been found, the block is reconstructed, decompressed (if it was compressed), its SHA-1 hash is verified and, in case of successful verification, returned to the user.

In general, sequential reading is very efficient since the blocks are read in the same order that they were written and the individual fragments end up getting prefetched from disk into local memory. Usually, the requested fragment is present in the current component location. However in some cases (for example after intermittent failures), the requested fragment may only be present in one of the previous locations of this component. In such a case, the component directs a distributed search for the missing data. In particular, the trail of previous component locations can be searched in the reverse order.

3.3 Load Balancing

In a distributed storage system like the HYDRAsstor back-end, the distribution of data among physical nodes is critical for system survivability, data resiliency and availability, storage utilization, and system performance. For example, placing too many peer components on one machine may have catastrophic consequences if this node is lost. The affected supernode may not recover, because too many components have been lost; and even when it is recoverable, some or even all of the data handled by this supernode may not be readable, due to loss

of too many fragments. Also, performance of the system is maximized when components are assigned proportionally to available node resources, since the load on each node is proportional to the prefix space covered by the components assigned to this node.

Our system continuously attempts to balance component distribution over all physical machines to reach a state where failure resiliency, performance and storage utilization are maximized. The quality of a given distribution is measured by a multi-dimensional function prioritizing these objectives, called *system entropy*. Balancing is carried out by each machine, which periodically considers all possible transfers of locally hosted components to neighboring nodes. If the machine finds a transfer that would improve the distribution, it is executed. After a component arrives at a new location, its data is also moved from old location(s) to the new one; but this data transfer happens in the background and may take a long time.

The same entropy-driven balancing is applied to the system when nodes are added or removed from the system.

3.4 Impact of Supernode Cardinality

Selection of supernode cardinality has profound impact on properties of HYDRAsstor. First of all, it determines the maximal number of tolerated node failures. The network overlay, but not necessarily user data, survives node failures as long as each supernode remains alive. A supernode survives if at least half of the supernode's peers plus one remain alive so they can reach a consensus.

Supernode cardinality also influences scalability, at least in theory. For a given cardinality, the probability that each supernode survives is fixed; the higher the cardinality the higher the probability of survival. When a system size grows, its number of supernodes also grows, and, as a result, the system reliability decreases, as for the system to be operational we require all supernodes to be alive. However, the practical impact of this limitation is negligible in the target range of system size, because permanent loss of a physical node is very rare, and self-healing reduces the window of vulnerability even when it happens.

Finally, supernode cardinality defines the number of data redundancy classes available. Erasure coding is parametrized with the maximal number of fragments that can be lost while a block remains still reconstructible (standard m -of- n erasure codes with n set to supernode cardinality and m determined by the redundancy class; we use the Cauchy-based Reed-Solomon codes [9]). Since in HYDRAsstor the erasure coding always produces supernode cardinality fragments, the tolerated number of lost fragments can vary from one to supernode cardi-

nality minus one (in the latter case we keep supernode cardinality copies of such block). Each such choice of tolerated number of lost fragments defines one data redundancy class. Each class represents different tradeoff between storage overhead (due to erasure coding) and failure resiliency.

4 Data Organization

Proper representation of stored data is critical for meeting reliability, availability and performance targets of HYDRAsTOR. The system should be able to easily identify the availability of stored data, and in case of a failure, rebuild only the data actually written and only to the requested resiliency level (as opposed to RAID, which rebuilds entire disk even if it contains no valid user data). Since components move between nodes followed by the data transfer, it should be possible to locate and retrieve data from old component locations. When such data is available, it should be transferred instead of being rebuilt, as transfer is a much cheaper operation. Data written in one stream should be placed nearby to maximize write and read performance. Last but not least, the data organization should support on-demand distributed data deletion, in which data blocks not reachable from any live retention root are deleted and the space occupied by them is reclaimed.

4.1 Synchrans and Synchrans Components

As discussed earlier, we use erasure coding for data redundancy. Resulting fragments of one block are distributed to peer components of the supernode responsible for this block. The basic logical unit of data management in HYDRAsTOR is the *synchrans*, containing a limited number of blocks written consecutively by one write-handling peer component.

A synchrans is analogous to a stripe in a RAID group since both allow faster reads and writes of continuous data faster than any single disk can do. Unlike a RAID stripe, a synchrans is also the basic block that is used for data balancing and load management as described below. Since writing a block really means writing a supernode cardinality of its fragments, each synchrans is represented by supernode cardinality of *synchrans components*, one for each peer. For the i -th peer of a supernode, the corresponding synchrans component contains all i -th fragments of the synchrans blocks. A synchrans is a logical structure only, but synchrans components actually exist on corresponding peers.

4.2 Chains of Containers

At any given time, each write-handling peer writes block fragments to exactly one synchrans. As a result, all such synchrans can be logically ordered in a chain, with the order determined by the write-handling peer. Synchrans components are placed in a data structure called *synchrans component container (SCC)*. Each SCC can contain one or more chain-adjacent synchrans components, and as a result, SCCs form also chains similar to synchrans component chains.

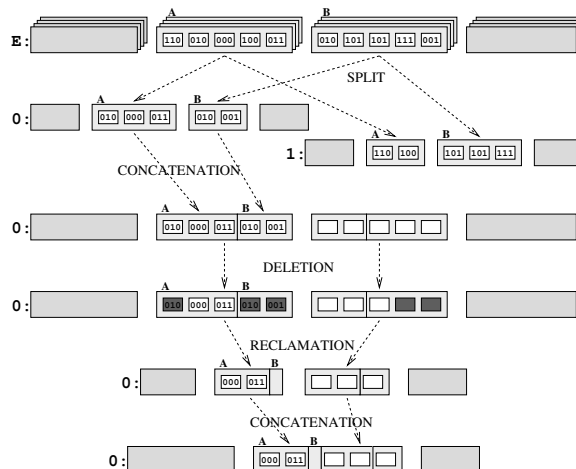


Figure 3: Data organization with synchrans and synchrans containers.

The upper row in Fig. 3 shows synchrans A and B that belong to the empty prefix supernode which covers the entire hashkey space. Each synchrans component is placed here in one SCC, with its individual fragments represented by smaller boxes inside the SCC. SCCs with synchrans components of these synchrans are shown as rectangles placed one behind the other. A chain of synchrans is represented by the supernode cardinality of SCC chains, we call them *peer SCC chains*. In the remainder of the Fig. 3 we show only one such peer SCC chain.

Peer SCC chains are normally identical with regards to the synchrans components' metadata and the number of fragments they hold, but there are occasional differences caused by node failures which cause holes in the chains. This chain organization allows for relatively simple and efficient implementation of required features. For example, if the number of peer chains without any holes is not lower than the number of fragments needed to reconstruct each block, then we infer that the data is available (i.e. all blocks are reconstructible). In such way, determination of data availability can easily be made for each redundancy class.

Each supernode will eventually be split as more data is stored or as more nodes are added to the system. This is

a regular FPN split which results in two new supernodes with prefixes extended from their ancestor prefix with, respectively, 0 and 1. After a supernode split, each synchronun in this supernode is also split, with fragments distributed between them based on their hash prefixes. The second row of Fig. 3 shows two such chains, one for the supernode with the prefix 0, and the other with the prefix 1. As a result of the split, fragments from synchronuns A and B are distributed to these two chains. The system now has 4 synchronuns, each approximately half the size of the originals.

The system strives to maintain a limited number of local SCCs, and merges adjacent synchronun components into one SCC (as shown on the third row of Fig. 3) until the maximum size of an SCC is reached. By limiting the number of local SCCs, the system can keep their metadata cached in RAM which enables fast determination of actions needed for providing data services. The target size of an SCC is a configuration constant (usually set well below 100 MB), so multiple SCCs can be read into the main memory. These SCC concatenations are loosely synchronized on all peers, so peer chains look the same. A similar operation is needed after deletion, shown in the remaining rows of this figure and discussed later in Section 5.3

This data organization is relatively simple in a static system, but it becomes quite complex due to the dynamic nature of the HYDRAsstor back-end. For example, when a peer is transferred to another physical node because of load balancing, its chains are transferred in the background to a new location, one SCC at a time. Similarly, after a supernode split, not all SCCs of the supernode are split immediately; instead we run background operations adjusting chains to the current supernode locations and shape. As a result, in any given moment, we may have chains partially-split, partially present in previous locations of this peer, or both. After failure, we may have serious holes in some of the chains. Fortunately, since peer chains describe the same data, we have supernode cardinality chain redundancy in the system, so usually there is a sufficient number of complete chains. This chain redundancy allows for reasoning about the data in the system even in the presence of transfers/failures. Additionally, more refined algorithms are used in some cases, constructing chain coverage from chain parts present on different peers.

5 Data Services

Based on the data organization described above, HYDRAsstor efficiently builds data services like identification of the recoverability of data, deletion and space reclamation, locating data in the network, data deduplication and others. Given a detailed description of all of

these features is beyond the scope of this paper, but this section will present a sketch of the data rebuilding, deletion and duplicate elimination services.

5.1 Data Rebuilding

When a node or disk fails, the SCC's residing on that node or disk are lost. As a result, the redundancy of the data blocks with fragments belonging to these SCCs is at best reduced below the level requested by users when writing these blocks. In the worst case, a block may be lost completely if not enough fragments survive. To keep the block redundancy at the desired levels, the system scans SCC chains looking for holes and schedules data rebuilding as background jobs for each missing SCC.

Multiple peer SCCs can be rebuilt in one rebuilding operation. Based on SCC metadata, the minimal number of peer SCCs needed for rebuilding is read by the peer that is in charge of rebuilding. This peer does bulk erasure decoding and encoding to restore the missing fragments. Next, the rebuilt SCCs are sent to the current target locations. Before SCCs are rebuilt, all input SCCs are made to look the same, i.e. required splits and concatenations are performed first. This requirement allows for fast bulk rebuilding as measured in Section 6.

5.2 Duplicate Elimination

Duplicate elimination can be classified in many dimensions: (1) the granularity of the deduplication: whole files, partial files, fixed size blocks or variable sized blocks; (2) time when the deduplication occurs: inline during the write phase or as a background process; (3) precision of duplicate identification: can the system reliably find all duplicates or does it use an approximate technique which trades precision for increased performance? (4) the verification of equality between a duplicate and its copy: just by comparison of hashes or with full data comparison; (5) the scope of the deduplication: the whole system (global deduplication), or the deduplication limited for example to data on a specific node (local deduplication).

Today HYDRAsstor implements variable-sized block, inline, hash-verified global duplicate elimination implemented on storage nodes. Variable-size blocks allow for better deduplication, because content-dependent chunking can be used ([40]). Inline deduplication increases write throughput, since duplicated block writes can be handled without writing to disk; this also increases storage efficiency compared to off-line deduplication. For regular blocks, we use fast approximate deduplication, whereas for retention roots, we do reliable duplicate elimination to ensure that searchable retention roots with the same search key but different contents are not written.

In both cases, for successful deduplication, we require that the potential duplicate of the block being written has a redundancy class not weaker than the class requested by this write and that the potential old duplicate is reconstructible.

On a regular block write, the peer handling this write is selected based on the hash of this block. It means that two identical blocks written when this peer is alive will be handled by it, and the second block will be found a duplicate of the first one.

A more complicated case arises when the write-handling peer has been recently created because of transfer or component recovery, and it does not have yet all the data it should have, i.e. its local SCC chain is not complete. In this case, we go to the longest-alive peer in the current supernode to check for possible duplicates. This is just a heuristics, as this peer may also not have the proper SCC chain complete, so a duplicate may not be detected. However, such a miss occurs only in corner cases, after massive failures when most likely all chains are broken. Moreover, for a particular block, we miss only one opportunity to eliminate a duplicate; the next duplicate block will be deduplicated unless another failure or transfer of this peer happens.

For retention roots, we need to ensure that two blocks with the same search key have identical contents (otherwise retention roots would not uniquely identify snapshots). As a result, we need accurate duplicate elimination for retention roots. When a local SCC chain has holes at the peer handling this write, the peer sends duplicate elimination queries to all other peers in this supernode. Each of these peers checks locally for a duplicate. A negative answer also includes a summary description of the parts of the SCC chain on which this answer is based. The write handling peer collects all replies. If there is at least one positive, a duplicate is found; otherwise, when all are negative, this peer tries to determine if SCC information attached to negative replies covers one entire SCC chain. If yes, the new block is not a duplicate; otherwise such determination cannot be done and the write is rejected with special error status indicating that data rebuilding is in progress (this may happen after massive failures); in such case this write should be submitted later. Needless to say, such situations so far happened only in special tests, and never in practice.

5.3 Deletion and Space Reclamation

Implementing data deletion in a system like HYDRAsTOR turned out to be surprisingly difficult because of many challenges which stem from the nature of the system: content-addressability, distribution, failure tolerance, and duplicate elimination. While deletion in our content-addressable system is somehow similar to dis-

tributed garbage collection [29], which is well understood, overcoming the remaining challenges, discussed below, required new research.

When deciding if a block is to be duplicate-eliminated against its older copy, we must be sure that this old block is not scheduled for deletion. Deciding which block to keep and which to delete must be globally consistent and robust in the presence of failures. For example, a deletion decision made should not be temporarily lost due to intermittent failures, as otherwise we may eliminate duplicates using blocks which are really scheduled for deletion. Moreover, the robustness of the data deletion algorithm should be higher than the data robustness. As a result, even if some blocks are lost, data deletion should be able to proceed to logically remove the lost data and heal the system if requested to do so by the user.

To simplify the design and make the implementation manageable, we have implemented deletion in two phases. During the first phase, the system is read-only and blocks are marked for deletion. In the second phase, the data can be read and written, as the system reclaims the blocks marked for deletion. Having a read-only phase simplified the deletion implementation, because such approach lets us eliminate the impact of writes on marking blocks for removal.

Deletion is implemented with a per-block reference counter that counts the number of pointers in blocks in the system pointing to this block. Reference counters are not updated immediately on write. Instead, they are updated later in the read-only phase processing all pointers written since the previous read-only phase (so the counter update is incremental). For each such pointer, the reference counter of the pointed block is incremented. After all such incrementation is completed, all blocks with reference counter equal to zero are marked for deletion (dark-shaded fragments in Fig. 3). Moreover, reference counters of blocks pointed by blocks already marked for deletion (including roots with associated deletion roots) are decremented. Next, the whole decrementation process (i.e. marking for removal blocks with reference counters equal to zero and decrementing reference counters of blocks pointed by pointers included in these blocks) is repeated, until no more new blocks can be marked for deletion. At this point, the read-only phase ends, and blocks marked for deletion can be removed in the background.

The deletion algorithm described above requires that the metadata of all blocks, as well as all the pointers, be present before proceeding. The pointers and block metadata are replicated on all peers, so the deletion can proceed even if some blocks are no longer reconstructible, as long as at least one block fragment exists.

Since blocks are really kept as fragments, a copy of the block reference counter is kept per-fragment, and each

fragment of a given block should have the same value of this counter. Reference counters are computed independently on peers participating in the read-only phase. Before deletion is started, each such peer must have its SCC chain complete with respect to fragment metadata and pointers. Not all peers in a supernode have to participate, but some minimal number of peers is required to complete the read-only phase. The computed counters are later propagated in the background to remaining peers.

The redundancy in counter computation allows a deletion decision to survive node failures. However, the intermediate results of deletion computations are not persistent. Any failure before the decision is made wipes out these results on the affected nodes, and the whole computation needs to be repeated if too many peers cannot participate in this phase any more. Deletion can still continue, if a sufficient number of peers in each supernode are not affected by the failure. Upon conclusion of the read-only phase, the new counter values are made failure-tolerant. All dead blocks i.e. blocks with counters equal to zero are then swept out from physical storage in the background (reclamation in Fig. 3). Free space fragmentation is avoided by rewriting the whole synchron component container, copying only fragments of live blocks to the new location.

6 Evaluation

Each current HYDRAsstor storage node (SN) runs one back-end server, and has six 500 GB SATA disks, 6GB RAM, two dual-core 3 GHz CPUs and two GigE cards. Some experiments have also been done with the experimental next generation hardware (denoted SN2), in which each storage node runs two back-end servers and has twelve 1 TB SATA disks, 20 GB of RAM, two quad-core 3GHz CPUs and four GigE cards. In all experiments, we used the current access node (AN) with 6GB RAM, two dual-core 3 GHz CPUs, two GigE cards and only limited local storage. All nodes run the Red Hat EL 5.1 version of Linux.

All experiments were performed using block size of 64KB compressible by 33% to 48KB except where noted. The system was configured with a supernode cardinality of 12 and the number of supernodes was equal to the number of physical machines. All of the tests wrote data using a resiliency class which has 9 original and 3 redundant fragments.

6.1 Read/Write Bandwidth

This experiment shows write throughput as a function of the fraction of blocks detected as duplicates for two

different compression ratios. We have used 4 SN2 machines, and 4 AN machines, each with one testing driver able to generate a stream of blocks with a specified percentage of duplicates and compression ratio. Duplicates are evenly distributed in the stream. Duplicated data is written in the same order as the base data, re-creating the original data stream. For the read experiment, the testing driver attempts to read data in the same order as it was written.

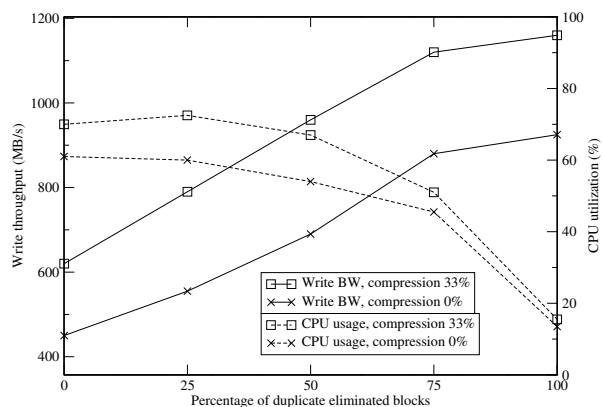


Figure 4: Write throughput as a function of duplicate ratio.

As shown in Fig. 4, very high bandwidth is achieved, which is a consequence of a carefully chosen data organization utilizing bulk transfer to disk. Duplicates are processed much more effectively than non-duplicated data, because they do not require fragmentation, fragment distribution and storage. Moreover, SCC-based organization allows the write-handling peer to perform fast local duplicate elimination by checking block reconstructibility with SCC reports submitted in the background from the remaining peers. However, when all writes are duplicates, the network bandwidth between AN and SNs becomes a bottle-neck, and the overall performance does not increase as much as expected (both curves flatten a bit at 100% duplicates). For high deduplication ratios, the CPU utilization decreases dramatically and network bandwidth between storage nodes remains available, so background tasks like data reconstruction and data scrubbing can be run without impact on user-visible performance.

Read bandwidth highly depends on factors like the sequentiality of the data read, the number of drivers reading simultaneously and the granularity of the distribution of the duplicates in the data. A detailed discussion of the impact of these factors on read performance is beyond the scope of this paper. Instead, we give read throughput achieved when reading the data written during the experiment described above. With four drivers reading, the total combined read bandwidth for indicated levels of deduplication was between 450 MB/s and 790MB/s

for 33% compressible data, and between 400 MB/s and 550MB/s for 0% compressible data.

The time required to fill the 4 SN2 node system depends on the percentage of duplicates in the data written. The system can be filled in 1 day when writing with no duplicates, while filling a system with 95% duplicated data can take up to 10 days. In general, for configurations in which high performance is not a priority, fewer ANs can be used resulting also in extended time-to-fill.

These results were obtained with testing drivers running on the ANs. Experiments with real backup applications using the filesystem front-end yielded similar performance. However, since the experiments were not done in a controlled setup, their results are not presented here.

6.2 System Scaling

This experiment, with up to 12 SNs and the number of ANs set to half of the number of SNs, shows how performance is scaled when numbers of storage nodes and access nodes increase. Two sets of measurements are done — a dynamic one, in which nodes are added while the user is writing, and a static one in which the number of nodes remains constant during the test. In the latter case, each measurement was taken after re-initializing the system from scratch and then loading the same amount of random, non-duplicated data. Time on the X axis refers to the dynamic case only.

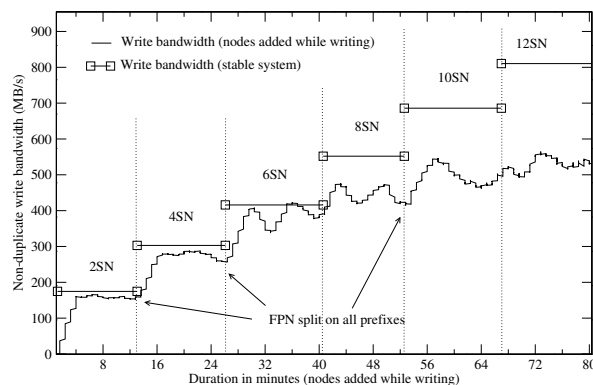


Figure 5: Dynamic vs. static scalability test.

The results indicate that in the range of nodes tested the system performance scales linearly with the system growth in the static case. The system attempts to balance components so that the hash space is divided equally across storage nodes. Such balancing guarantees that every machine is equally loaded and does not become a bottleneck. In the dynamic case, the cost of dynamically reconfiguring the system results in lower user bandwidth. This happens since most of the data is on the older nodes which are checked on every write for duplicate elimina-

tion. However, after all data transfers are completed, the performance in the dynamic case will be the same as in the stable case.

6.3 Node Failure and Data Rebuilding

This experiment shows the system behavior and its performance just after node failure, during resulting data reconstruction, and after the failed node is recovered. The system tested has 4 SN2 machines and 4 AN machines.

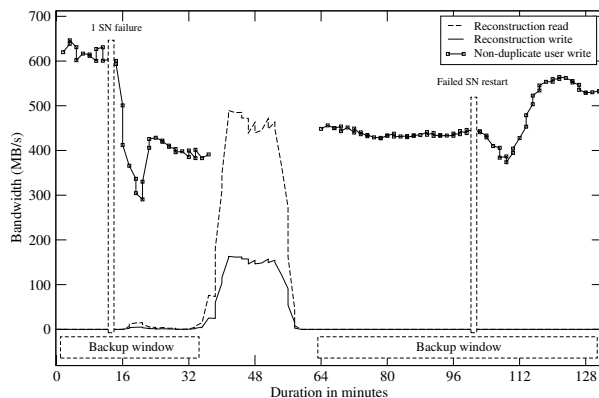


Figure 6: Node failure during backup.

We started writing to the healthy system with four storage nodes, achieving write throughput over 600 MB/s. After about 14 minutes one storage node failed (both back-end servers crashed). Write performance just after the node failure dropped to 300 MB/s, then stabilized at about 400 MB/s. The initial drop was caused by timeout messages to the failed node and overhead for system rebalancing. Data rebuilding (reconstruction) tasks were ordered, however they were suppressed because of the ongoing user backup. Reconstruction started to work with full bandwidth just after all user writes had been finished. Every block reconstruction required reading all 9 remaining fragments in order to rebuild the 3 lost ones. The reconstruction read bandwidth reached 480 MB/s on the 3 surviving machines with a reconstruction write bandwidth of 160 MB/s. The rebuilding finished in the 58th minute of the experiment leaving a healthy system with only 3 storage nodes.

In the 64th minute the next writing session started achieving write bandwidth of 430 MB/s. The failed node was recovered and connected once again in the 100th minute. Just after the re-connection, system write bandwidth dropped to 380 MB/s, but when components rebalancing was finished it increased to about 550 MB/s. At the end of the experiment the system had 4 storage nodes, however it was not healthy, as not all data (SCCs) were in the correct places. Write performance will increase to the initial (600 MB/s) after all pending transfers are finished and the system becomes healthy again.

The results show that the system maximizes user bandwidth during backup even if background tasks are pending. In particular, ongoing reconstruction is suspended if a new backup is started. This approach allows a user to minimize costly backup windows regardless of internal system state, but carries the risk of starvation of critical data rebuilding tasks. However, this may happen only if the system is fully loaded by a user all the time and only when the user writes non-duplicated data. If the user load decreases or some duplicates are written, reconstruction is executed in the background. Finally, this experiment also shows how quickly the system adjusts to changes in its environment, as it takes only a few minutes for the system to fully utilize released resources.

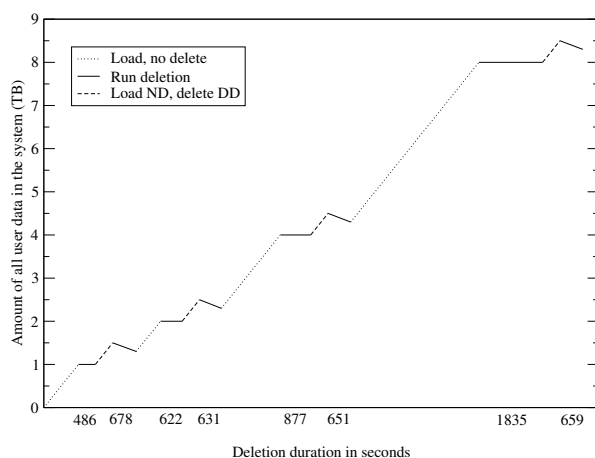


Figure 7: Read-only phase duration.

6.4 Data Deletion

The purpose of this experiment is to evaluate the duration of the read-only phase as a function of the data loaded with a focus on scenarios reflecting a realistic system usage. Therefore, we use the file system interface to write and delete data periodically, increasing overall the amount of data stored in the system. Deletion experiments were performed using 4 *SN2* machines and 1 *AN* machine.

The test runs the following four step sequence four times. During the initial step (shown with the dotted lines in the Fig. 7), the data is loaded into the system. The loading phase pauses at 1TB, 2TB, 4TB and 8TB. In the second step, during each pause, the read-only data deletion phase is run which recomputes the counters for the newly loaded data (note that no data is marked for deletion at this point). The duration of this step is shown with light-gray bars. The third step (shown with dashed lines), an additional half terabyte of new data (ND) is loaded and a user invokes a deletion operation of 0.2 TB of older data (DD). In the last step, one more read-only

phase is run to recompute the counters to reflect the recently loaded data and mark the blocks for deletion.

The duration of each read-only phase is shown with dark-gray bars. In all cases, the new data is not compressible and does not contain any duplicates, but with duplicates present the results will be similar, except that all phases will be shorter.

Although the X axis in Fig. 7 shows duration of each read-only phase, the data-loading steps are not shown in proportion there, because they are too big (we load terabytes of data and it takes several hours). We note that all read-only phases are relatively short, the longest one, after loading 3.7 TB of data (which took about 4 hours) is about 30 minutes, resulting in deletion time of under 13% of writing time. For writing with two ANs, this fraction can go up to 20% in case of not-duplicated streams. When writing data with a high number of duplicates (the common case with backups), deletion takes significantly less time (on the order of 5% of the writing time), since less data needs to be read to access the pointers, and filling the capacity takes so much longer. Moreover, the duration of the first read-only phase (shown with the light-gray bars) in each sequence is proportional to the new data loaded in the first step of the scenario. Finally, the duration of the second read-only phase (shown with dark bars) is fairly constant, taking around 11 minutes per run. This also shows the power of the incremental reference-counting deletion in HYDRAsstor. The duration of the read-only phase depends only on the amount of data added and deleted since the previous run of this phase, but not on the total amount of data in the system.

7 Related Work

A significant number of distributed storage systems [8, 10, 11] are designed as large scale systems which are distributed over wide area networks and built with untrusted peers. These systems undergo frequent configuration changes. For example, the goal of OceanStore [8] was to provide reliable storage for all data ever created. These systems concentrated on scalability (e.g. OceanStore, PAST [11]) and tolerating a large class of failures, including Byzantine and large-scale correlated failures (Glacier [18]) at the expense of performance.

Another group of distributed storage systems targeted the data center and, in this, are more like HYDRAsstor. These systems include distributed virtual disk like Petal [19], distributed file systems like CEPH [37] and Farsite [6], clustered file systems like Sorrento [34], Panasas [39], and GoogleFS [15], clustered storage including Ursa Minor [5], RADOS [38], and FAB [28]. Compared to HYDRAsstor these systems have different target applications and are not advertised as secondary storage. As a result, they do not provide deduplication

(except Farsite, which does it on file level); these systems are not CAS-based, but need to deal with issues of consistency in the presence of write-sharing, which do not occur in our system. Ursa Minor does support user-selected choices of data resiliency, similar to our data resiliency classes. DISP [14] is a flexible system that can be specialized to both WAN and data center. Like HYDRAstor, DISP uses erasure codes, but it does not provide deduplication.

Venti [24], EMC Centera [4, 16], Pergamum [32] and DataDomain [43] are secondary storage systems. Venti, Pergamum and Centera target archiving, whereas DataDomain is designed to store backup data. Pergamum does not support duplicate elimination, Venti prototype and Centera do it, respectively, on fixed block size and entire file level. Centera might be able to do chunk-level deduplication, based on available information [16], but the chunk size seems to be much larger (100MB per chunk versus the HYDRAstor 64KB). These approaches result in lower deduplication than a variable-block size approach used by HYDRAstor and DataDomain. However, DataDomain is a centralized system and does not do global deduplication in distributed environment. HYDRAstor provides global deduplication using also variable block chunking with comparable write performance. RepStore [41], a smart-brick scalable storage system, uses erasure codes and content-based addressing, but does not provide deduplication. Deep Store [40], an archiving system, employs multitude of techniques for reducing stored data size, including delta compression and variable-block-size deduplication. However, this system does not target backup data.

Blocks in our system have some resemblance to objects in the object-based storage [22], as they have attributes (for example resiliency class) and simple interface to access its components like pointers.

Many systems introduce structures similar to SCCs for block aggregation. Venti uses arenas to serve as a unit of data maintenance; however, they do not take advantage of the sequential nature of incoming data streams and achieve very low performance. The Foundation [26] CAS Layer improves Venti's sequential write performance by prefetching entire arenas when duplicates are written. However, since Foundation is designed for personal use, it does not have to deal with the problem of multiple streams written concurrently but later read separately. DataDomain introduces containers to group sequential writes from each stream of data to increase effectiveness of read-ahead caching. HYDRAstor achieves a similar result by sorting incoming blocks by their stream id and flushing them out to disk in batches. Using separate containers for every stream in HYDRAstor is not feasible, as the number of containers written concurrently may be very large for big systems. HYDRAstor

data organization is unique in use of replicated chains of containers which allow for reasoning about state of the data in the system.

Deletion in a distributed storage system is relatively simple if there is no duplicate elimination. It can be done with leases like in Glacier [18], or with simple reclamation of obsolete versions like in Ursa Minor. However, with deduplication, deletion becomes difficult for reasons explained earlier. For example, Venti and Deep Store have not implemented deletion. As far as we know, the HYDRAstor back-end approach to deletion is unique. The use of blocks with pointers, retention and deletion roots and redundant chains of containers enables an efficient, fault-tolerant implementation of a distributed deletion.

8 Conclusions and Future Work

HYDRAstor is a decentralized, scalable secondary storage that is commercially available today. It can be used as an on-line repository for all enterprise backup and archival data while dynamically and efficiently sharing available capacity. Critical features like high-availability and reliability, ease of management, capacity and performance scalability, and storage efficiency make the system unique in addressing today's enterprise needs. The system is externally visible as one storage pool and can be accessed by legacy applications using traditional file system interface.

The core architecture is built around a DHT with virtual supernodes spanned over physical nodes. Data resiliency is provided with erasure codes, with fragments of erasure-coded blocks distributed among supernode components. Redundancy in the network and data allows for on-line upgrades and extensions, increasing availability of the system. High storage efficiency is facilitated by variable block size global deduplication. The back-end exports a low-level API providing operations on content-addressed blocks which expose pointers to other blocks. A novel data organization based on redundant chains of data containers is used to deliver reliably multitude of data services, including failure-tolerant deletion and fast verification of data health.

Although the system is fully functional today, there is an important work left to improve its value delivered to the end user. The read-only phase of deletion will be eliminated, which will make the system fully usable all the time. Deduplication can be moved to a proxy server, saving bandwidth and improving write performance of highly-duplicated streams. Additionally, since multiple types of drivers can write to the back-end, there is a need for a stream interface that can cut data into blocks in a standard way. This will ensure higher deduplication among data written by different types of clients.

References

- [1] Plasmon and Pegasus. Archival Storage Total Cost of Ownership Analysis, 2005. <http://www.pegasus-afs.com/PDFs/WhitePapers/ArchiveStorageTCOReport.pdf>.
- [2] Vtf open, 2005. <http://www.diligent.com/products:VTF-Open-2>.
- [3] Overland storage unveils reo 9500d all-in-one deduplicating vtl appliance, 2007. <http://www.overlandstorage.com>.
- [4] EMC Corp. EMC Centera: content addressed storage system, 2008. <http://www.emc.com/products/family/emc-centera-family.htm?openfolder=platform>.
- [5] ABD-EL-MALEK, M., II, W. V. C., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M. P., PRASAD, M., SALMON, B., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Ursa minor: Versatile cluster-based storage. In *FAST* (2005).
- [6] ADYA, A., BOLOSKEY, W., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J., HOWELL, J., LORCH, J., THEIMER, M., AND WATTENHOFFER, R. Farsite: Federated, available, and reliable storage for an incompletely trusted environment, 2002.
- [7] AJMANI, S., LISKOV, B., AND SHRIRA, L. Modular software upgrades for distributed systems. In *European Conference on Object-Oriented Programming (ECOOP)* (July 2006).
- [8] BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., ZHAO, B., AND KUBIATOWICZ, J. Oceanstore: An extremely wide-area storage system. Tech. rep., Berkeley, CA, USA, 1999.
- [9] BLÖMER, J., KALFANE, M., KARPINSKI, M., KARP, R., LUBY, M., AND ZUCKERMAN, D. An xor-based erasure-resilient coding scheme. Tech. Rep. TR-95-048, International Computer Science Institute, August 1995.
- [10] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with cfs. *SIGOPS Oper. Syst. Rev.* 35, 5 (2001), 202–215.
- [11] DRUSCHEL, P., AND ROWSTRON, A. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII* (Schloss Elmau, Germany, May 2001), pp. 75–80.
- [12] DUBNICKI, C., UNGUREANU, C., AND KILIAN, W. FPN: A Distributed Hash Table for Commercial Applications. In *Proceedings of the Thirteenth International Symposium on High-Performance Distributed Computing (HPDC-13 2004)* (Honolulu, Hawaii, June 2004), pp. 120–128.
- [13] EL MALEK, M. A., II, W. V. C., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M., PRASAD, M., SALMON, O., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Early experiences on the journey towards self-* storage. *IEEE Data Eng. Bulletin* (2006).
- [14] ELLARD, D., AND MEGQUIER, J. Disp: Practical, efficient, secure and fault-tolerant distributed data storage. *Trans. Storage* 1, 1 (2005), 71–94.
- [15] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.
- [16] GUNAWI, H. S., AGRAWAL, N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SCHINDLER, J. Deconstructing Commodity Storage Clusters. In *Proceedings of the 32nd International Symposium on Computer Architecture* (Madison, WI, June 2005).
- [17] GUPTA, S. D. Network Magazine. Addressing Storage Management Challenges, 2002. <http://www.networkmagazineindia.com/200212/cover1.shtml>.
- [18] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: highly durable, decentralized storage despite massive correlated failures. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2005), USENIX Association, pp. 143–158.
- [19] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1996), pp. 84–92.
- [20] MAYMOUNKOV, P., AND MAZIERES, D. Kademlia: A peer-to-peer information system based on the xor metric. In *In Proceedings of IPTPS02* (Cambridge, USA, March 2002).
- [21] MERRILL, D. Hitachi Data Systes. Storage Economics: Identifying and Reducing Operating Expenses in the Storage Infrastructure, 2003. <http://www.hds.com/pdf/StorageEconomicsWHP-153.pdf>.
- [22] MESNIER, M., GANGER, G. R., AND RIEDEL, E. Object-based storage. *IEEE Communications Magazine* 41 (2003), 84–90.
- [23] NEC Corporation. HYDRAsstor Grid Storage System, 2008. <http://www.hydrastor.com>.
- [24] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies* (Monterey, CA, 2002).
- [25] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2001), ACM, pp. 161–172.
- [26] RHEA, S., COX, R., AND PESTEREV, A. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the 2008 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2008), USENIX Association, pp. 143–156.
- [27] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science* 2218 (2001), 329+.
- [28] SAITO, Y., FROLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. Fab: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.* 38, 5 (2004), 48–58.
- [29] SHAPIRO, M. A survey of distributed garbage collection techniques. In *In Proceedings of the 1995 International Workshop on Memory Management* (1995), Springer-Verlag, pp. 211–249.
- [30] SOULES, C. A. N., APPAVOO, J., HUI, K., WISNIEWSKI, R. W., SILVA, D. D., GANGER, G. R., KRIEGER, O., STUMM, M., AUSLANDER, M., OSTROWSKI, M., ROSENBERG, B., AND XENIDIS, J. System support for online reconfiguration, June 2003.
- [31] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications* (2001), ACM Press, pp. 149–160.
- [32] STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. Pergamum: replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–16.
- [33] STRUNK, J. D., AND GANGER, G. R. A human organization analogy for self-* systems. In *In First Workshop on Algorithms and Architectures for Self-Managing Systems, in conjunction with Federated Computing Research Conference* (2003), pp. 1–6.

- [34] TANG, H., GULBEDEN, A., ZHOU, J., STRATHEARN, W., YANG, T., AND CHU, L. A self-organizing storage cluster for parallel data-intensive applications. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2004), IEEE Computer Society, p. 52.
- [35] THERESKA, E., SALMON, O., STRUNK, J., WACHS, M., EL MALEK, M. A., LOPEZ, J., AND GANGER, G. R. Stardust: Tracking activity in a distributed storage system. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (Saint-Malo (2006))*, ACM Press, pp. 3–14.
- [36] WEATHERSPOON, H., AND KUBIATOWICZ, J. Erasure coding vs. replication: A quantitative comparison. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, 2002), Springer-Verlag, pp. 328–338.
- [37] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *OSDI'06: 7th USENIX Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 307–320.
- [38] WEIL, S. A., LEUNG, A. W., BRANDT, S. A., AND MALTZAHN, C. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *PDSW (2007)*, G. A. Gibson, Ed., ACM Press, pp. 35–44.
- [39] WELCH, B., UNANGST, M., ABBASI, Z., GIBSON, G., AND MUELLER, B. Scalable performance of the panasas parallel file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), USENIX Association, pp. 17–33.
- [40] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep store: An archival storage system architecture. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 804–8015.
- [41] ZHANG, Z., LIN, S., LIAN, Q., AND JIN, C. Repstore: A self-managing and self-tuning storage backend with smart bricks. In *ICAC (2004)*, pp. 122–129.
- [42] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications* 22 (2004), 41–53.
- [43] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–14.

Smoke and Mirrors: Reflecting Files at a Geographically Remote Location Without Loss of Performance

Hakim Weatherspoon, Lakshmi Ganesh, Tudor Marian, [†]Mahesh Balakrishnan, and Ken Birman
Cornell University, Computer Science Department, Ithaca, NY 14853

{hweather, lakshmi, tudorm, ken}@cs.cornell.edu

[†]*Microsoft Research, Silicon Valley*

maheshba@microsoft.com

Abstract

The Smoke and Mirrors File System (SMFS) mirrors files at geographically remote datacenter locations with negligible impact on file system performance at the primary site, and minimal degradation as a function of link latency. It accomplishes this goal using wide-area links that run at extremely high speeds, but have long round-trip-time latencies—a combination of properties that poses problems for traditional mirroring solutions. In addition to its raw speed, SMFS maintains good synchronization: should the primary site become completely unavailable, the system minimizes loss of work, even for applications that simultaneously update groups of files. We present the SMFS design, then evaluate the system on Emulab and the Cornell National Lambda Rail (NLR) Ring testbed. Intended applications include wide-area file sharing and remote backup for disaster recovery.

1 Introduction

Securing data from large-scale disasters is important, especially for critical enterprises such as major banks, brokerages, and other service providers. Data loss can be catastrophic for any company — Gartner estimates that 40% of enterprises that experience a disaster (e.g. loss of a site) go out of business within five years [41]. Data loss failure in a large bank can have much greater consequences with potentially global implications.

Accordingly, many organizations are looking at dedicated high-speed optical links as a disaster tolerance option: they hope to continuously mirror vital data at remote locations, ensuring safety from geographically localized failures such as those caused by natural disasters or other calamities. However, taking advantage of this new capability in the wide-area has been a challenge; existing mirroring solutions are highly latency sensitive [19]. As a result, many critical enterprises operate at risk of catastrophic data loss [22].

The central trade-off involves balancing safety against

performance. So-called synchronous mirroring solutions [6, 12] block applications until data is safely mirrored at the remote location: the primary site waits for an acknowledgment from the remote site before allowing the application to continue executing. These are very safe, but extremely sensitive to link latency. Semi-synchronous mirroring solutions [12, 42] allow the application to continue executing once data has been written to a local disk; the updates are transmitted as soon as possible, but data can still be lost if disaster strikes. The end of the spectrum is fully asynchronous: not only does the application resume as soon as the data is written locally, but updates are also batched and may be transmitted periodically, for instance every thirty minutes [6, 12, 19, 31]. These solutions perform best, but have the weakest safety guarantees.

Today, most enterprises primarily use asynchronous or semi-synchronous remote mirroring solutions over the wide-area, despite the significant risks posed by such a stance. Their applications simply cannot tolerate the performance degradation of synchronous solutions [22]. The US Treasury Department and the Finance Sector Technology Consortium have identified the creation of new options as a top priority for the community [30].

In this paper, we explore a new mirroring option called *network-sync*, which potentially offers stronger guarantees on data reliability than semi-synchronous and asynchronous solutions while retaining their performance. It is designed around two principles. First, it proactively adds redundancy at the network level to transmitted data. Second, it exposes the level of in-network redundancy added for any sent data via feedback notifications. Proactive redundancy allows for reliable transmission with latency and jitter independent of the length of the link, a property critical for long-distance mirroring. Feedback makes it possible for a file system (or other applications) to respond to clients as soon as enough recovery data has been transmitted to ensure that the desired safety level has been reached. Figure 1 illustrates this idea.

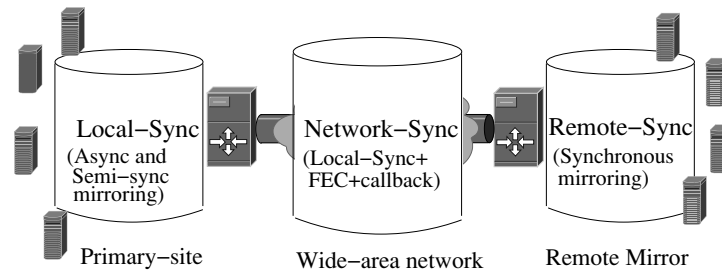


Figure 1: Remote Mirroring Options. (1) Synchronous mirroring provides a *remote-sync* guarantee: data is not lost in the event of disaster, but performance is extremely sensitive to the distance between sites. (2) Asynchronous and semi-synchronous mirroring give a *local-sync* guarantee: performance is independent of distance between mirrors, but can suffer significant data loss when disaster strikes. (3) A new *network-sync* mirroring option with performance similar to local-sync protocols, but with improved reliability.

Of course, data can still be lost; network-sync is not as safe as a synchronous solution. If the primary site fails and the wide-area network simultaneously partitions, data will still be lost. Such scenarios are uncommon, however. Network-sync offers the developer a valuable new option for trading data reliability against performance.

Although this paper focuses on the Smoke and Mirrors File System (SMFS), we believe that many kinds of applications could benefit from a network-sync option. These include other kinds of storage systems where remote mirroring is performed by a disk array (e.g. [12]), a storage area network (e.g. [19]), or a more traditional file server (e.g. [31]). Network-sync might also be valuable in transactional databases that stream update logs from a primary site to a backup, or to other kinds of fault-tolerant services.

Beyond its use of the network-sync option, SMFS has a second interesting property. Many applications update files in groups, and in such cases, if even one of the files in a group is out of date, the whole group may be useless (Seneca [19] calls this atomic, in-order asynchronous batched commits; SnapMirror [31] offers a similar capability). SMFS addresses the need in two ways. First, if an application updates multiple files in a short period of time, the updates will reach the remote site with minimal temporal skew. Second, SMFS maintains group-mirroring consistency, in which files in the same file system can be updated as a group in a single operation where the group of updates will all be reflected by the remote mirror site atomically, either all or none.

In summary, our paper makes the following contributions:

- We propose a new remote mirroring option called network-sync in which error-correction packets are proactively transmitted, and link-state is exposed through a callback interface.
- We describe the implementation and evaluation

of SMFS, a new mirroring file system that supports both capabilities, using an emulated wide-area network (Emulab [40]) and the Cornell National Lambda Rail (NLR) Ring testbed [1]. This evaluation shows that SMFS:

- Can be tuned to lose little or no data in the event of a rolling disaster.
- Supports high update throughput, masking wide-area latency between the primary site and the mirror.
- Minimizes jitter when files are updated in short periods of time.
- We show that SMFS has good group-update performance and suggest that this represents a benefit to using a log-structured file architecture in remote mirroring.

The rest of this paper is structured as follows. We discuss our fault model in Section 2. In Section 3, we describe the network-sync option. We describe the SMFS protocols that interact with the network-sync option in Section 4. In Section 5, we evaluate the design and implementation. Finally, Section 6 describes related work and Section 7 concludes.

2 What's the Worst that Could Happen?

We argue that our work responds to a serious imperative confronted by the financial community (as well as by other critical infrastructure providers). As noted above, today many enterprises opt to use asynchronous or semi-synchronous remote mirroring solutions despite the risks they pose, because synchronous solutions are perceived as prohibitively expensive in terms of performance [22]. In effect, these enterprises have concluded that there simply is no way to maintain a backup at geographically re-

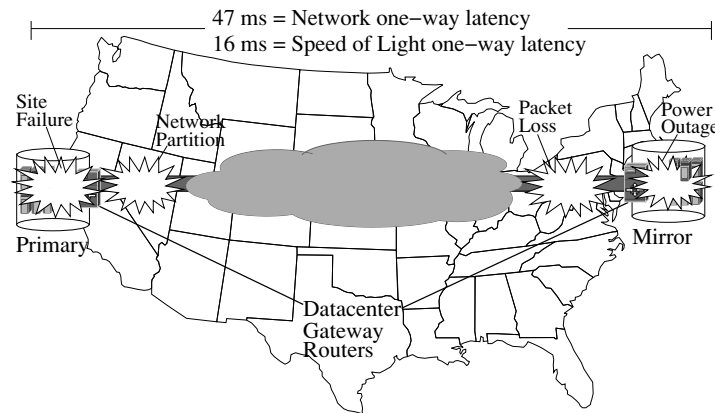


Figure 2: Example Failure Events. A single failure event may not result in loss of data. However, multiple nearly-simultaneous failure events (i.e. rolling disaster) may result in data loss for asynchronous and semi-synchronous remote mirroring.

mote distances at the update rates seen within their datacenters. Faced with this apparent impossibility, they literally risk disaster.

It is not feasible to simply legislate a solution, because today's technical options are inadequate. Financial systems are under huge competitive pressure to support enormous transaction rates, and as the clearing time for transactions continues to diminish towards immediate settlement, the amounts of money at risk from even a small loss of data will continue to rise [20]. Asking a bank to operate in slow-motion so as to continuously and synchronously maintain a remote mirrored backup is just not practical: the institution would fail for reasons of non-competitiveness.

Our work cannot completely eliminate this problem: for the largest transactions, synchronous mirroring (or some other means of guaranteeing that data will survive any possible outage) will remain necessary. Nonetheless, we believe that there may be a very large class of applications with intermediary data stability needs. If we can reduce the window of vulnerability significantly, our hypothesis is that even in a true disaster that takes the primary site offline and simultaneously disrupts the network, the challenges of restarting using the backup will be reduced. Institutions betting on network-sync would still be making a bet, but we believe the bet is a much less extreme one, and much easier to justify.

Failure Model and Assumptions: We assume that failures can occur at any level — including storage devices, storage area network, network links, switches, hubs, wide-area network, and/or an entire site. Further, we assume that they can fail simultaneously or even in sequence: a rolling disaster. However, we assume that the storage system at each site is capable of tolerating and recovering from all but the most extreme local failures. Also, sites may have redundant network paths con-

necting them. This allows us to focus on the tolerance of failures that disable an entire site, and on combinations of failures such as the loss of both an entire site and the network connecting it to the backup (what we call a rolling disaster). Figure 2 illustrates some points of failure.

With respect to wide-area optical links, we assume that even though industry standards essentially preclude data loss on the links themselves, wide-area connections include layers of electronics: routers, gateways, firewalls, etc. These components can and do drop packets, and at very high data rates, so can the operating system on the destination machine to which data is being sent. Accordingly, our model assumes wide-area networks with high data rates (10 to 40 Gbits) but sporadic packet loss, potentially bursty. The packet loss model used in our experiments is based on actual observations of TeraGrid, a scientific data network that links scientific supercomputing centers and has precisely these characteristics. In particular, Balakrishnan et al. [10] cite loss rates over 0.1% at times on uncongested optical-link paths between supercomputing centers. As a result, we emulate disaster with up to 1% loss rates in our evaluation of Section 5.

Of course, reliable transmission protocols such as TCP are typically used to communicate updates and acknowledgments between sites. Nonetheless, under our assumptions, a lost packet may prevent later received packets from being delivered to the mirrored storage system. The problem is that once the primary site has failed, there may be no way to recover a lost packet, and because TCP is sequenced, all data sent after the lost packet will be discarded in such situations — the gap prevents their delivery.

Data Loss Model: We consider data to be *lost* if an update has been acknowledged to the client, but the corresponding data no longer exists in the system. Today's remote mirroring regimes all experience data loss, but

the degree of disaster needed to trigger loss varies:

- Synchronous mirroring only sends acknowledgments to the client after receiving a response from the mirror. Data cannot be lost unless *both* primary and mirror sites fail.
- Semi-synchronous mirroring sends acknowledgments to the client after data written is locally stored at the primary site and an update is sent to the mirror. This scheme does not lose data unless the primary site fails *and* sent packets do not make it to the mirror. For example, packets may be lost while resident in local buffers and before being sent on the wire, the network may experience packet loss, partition, or components may fail at the mirror.
- Asynchronous mirroring sends acknowledgments to the client immediately after data is written locally. Data loss can occur even if just the primary site fails. Many products form snapshots periodically, for example, every twenty minutes [19, 31]. Twenty minutes of data could thus be lost if a failure disrupts snapshot transmission.

Goals: Our work can be understood as an enhancement of the semi-synchronous style of mirroring. The basic idea is to ensure that once a packet has been sent, the likelihood that it will be lost is as low as possible. We do this by sending error recovery data along with the packet and informing the sending application when error recovery has been sent. Further, by exposing link state, an error correcting coding scheme can be tuned to better match the characteristics observed in existing high-speed wide-area networks.

3 Network-Sync Remote Mirroring

Network-sync strikes a balance between performance and reliability, offering similar performance as semi-synchronous solutions, but with increased reliability. We use a forward-error correction protocol to increase the reliability of high-quality optical links. For example, a link that drops one out of every 1 trillion bits or 125 million 1 KB packets (this is the maximum error threshold beyond which current carrier-grade optical equipment shuts down) can be pushed into losing less than 1 out of every 10^{16} packets by the simple expedient of sending each packet twice — a figure that begins to approach disk reliability levels [7, 15]. By adding a callback when error recovery data has been sent, we can permit the application to resume execution once these encoded packets are sent, in effect treating the wide-area link as a kind of network disk. In this case, data is temporarily “stored” in the network while being shipped across the wide-area to the remote mirror. Figure 1 illustrates this capability.

One can imagine many ways of implementing this behavior (e.g. datacenter gateway routers). In general, implementations of network-sync remote mirroring must satisfy two requirements. First, they should *proactively* enhance the reliability of the network, sending recovery data without waiting for any form of negative acknowledgment (e.g. TCP fast retransmit) or timeouts keyed to the round-trip-time (RTT) to the remote site. Second, they must *expose* the status of outgoing data, so that the sender can resume activity as soon as a desired level of in-flight redundancy has been achieved for pending updates. Section 3.1 discusses the network-sync option, Section 3.2 discusses an implementation of it, and Section 3.3 discusses its tolerance to disaster.

3.1 Network-Sync Option

Assuming that an external client interacts with a primary site and the primary site implements some higher level remote mirroring protocol, network-sync enhances that remote mirroring protocol as follows. First, a host located at the primary site submits a write request to a local storage system such as a disk array (e.g. [12]), storage area network (e.g. [19]), or file server (e.g. [31]). The local storage system simultaneously applies the requested operation to its local storage image and uses a reliable transport protocol such as TCP to forward the request to a storage system located at the remote mirror. To implement the network-sync option, an egress router located at the primary site forwards the IP packets associated with the request, sends additional error correcting packets to an ingress router located at the remote site, and then performs a callback, notifying the local storage system which of the pending updates are now safely in transit¹. The local storage system then replies to the requesting host, which can advance to any subsequent dependent operations. We assume that ingress and egress routers are under the control of site operators, thus can be modified to implement network-sync functionality.

Later, perhaps 50ms or so may elapse before the remote mirror storage system receives the mirrored request—possibly after the network-sync layer has reconstructed one or more lost packets using the combination of data received and error-recovery packets received. It applies the request to its local storage image, generates a storage level acknowledgment, and sends a response. Finally, when the primary storage system receives the response, perhaps 100ms later, it knows with certainty that the request has been mirrored and can garbage collect any remaining state (e.g. [19]). Notice that if a client requires the stronger assurances of a true remote-sync, the possibility exists of offering that guarantee selectively, on a per-operation basis. Figure 3 illustrates the network-sync mirroring option and Table 1 contrasts it to existing solutions.

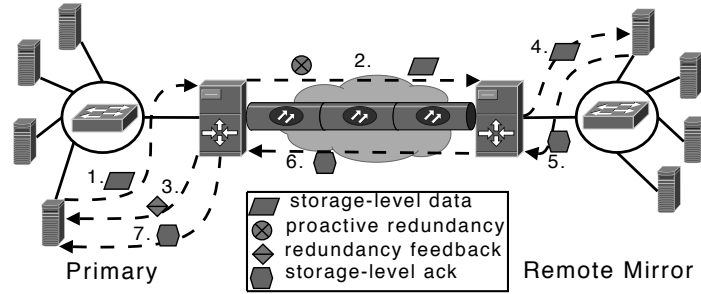


Figure 3: Network-Sync Remote Mirroring Option. (1) A primary-site storage system simultaneously applies a request locally and forwards it to the remote mirror. After the network-sync layer (2) routes the request and sends additional error correcting packets, it (3) sends an acknowledgment to the local storage system — at this point, the storage system and application can safely move to the next operation. Later, (4) a remote mirror storage system receives the mirrored request—possibly after the network-sync layer recovered some lost packets. It applies the request to its local storage image, generates a storage level acknowledgment, and (5) sends a response. Finally, (7) when the primary storage system receives the response, it knows with certainty that the request has been mirrored and can garbage collect.

Mirror Solution	Mirror Update	Mirror-ack Receive	Mirror-ack Latency	Rolling Disaster			
				Local-only Failure	Local + Pckt Loss	Local + NW Partition	Local+Mirror Failure
Local-sync	Async- or Semi-sync	N/A	N/A	Loss	Loss	Loss	Loss
Remote-sync	Synchronous	Storage-level ack (7)	WAN RTT	No loss	No loss	No loss	Maybe loss
Network-sync	Semi-sync	nw-sync feedback (3)	\approx Local ping	\approx No loss	\approx No loss	Loss	Loss

Table 1: Comparison of Mirroring Protocols.

3.2 Maelstrom: Network-sync Implementation

The network-sync implementation used in our work is based on Forward Error Correction (FEC). FEC is a generic term for a broad collection of techniques aimed at proactively recovering from packet loss or corruption. FEC implementations for data generated in real-time are typically parameterized by a rate (r, c) : for every r data packets, c error correction packets are introduced into the stream. Of importance here is the fact that FEC performance is independent of link length (except to the extent that loss rates may be length-dependent).

The specific FEC protocol we worked with is called Maelstrom [10], and is designed to match the observed loss properties of multi-hop wide-area networks such as TeraGrid. Maelstrom is a symmetric network appliance that resides between the datacenter and the wide-area link, much like a NAT box. The solution is completely transparent to applications using it, and employs a mixture of technologies: routing tricks to conceal itself from the endpoints, a link-layer reliability protocol (currently TCP), and a novel FEC encoding called *layered interleaving*, designed for data transfer over long-haul links with potentially bursty loss patterns. To minimize the rate-sensitivity of traditional FEC solutions, Mael-

strom aggregates all data flowing between the primary and backup sites and operates on the resulting high-speed stream. See Balakrishnan et al. [10] for a detailed description of layered interleaving and analysis of its performance tolerance to random and bursty loss.

Maelstrom also adds feedback notification *callbacks*. Every time Maelstrom transmits a FEC packet, it also issues a callback. The local storage system then employs a redundancy model to infer the level of safety associated with in-flight data packets. For this purpose, a local storage system needs to know the underlying network's properties — loss rate, burst length, etc. It uses these to model the behavior of Maelstrom mathematically [10], and then makes worst-case assumptions about network loss to arrive at the needed estimate of the risk of data loss. We expect system operators monitor network behavior and periodically adjust Maelstrom parameters to adapt to any changes in the network characteristics.

There are cases in which the Maelstrom FEC protocol is unable to repair the loss (this can only occur if several packets are lost, and in specific patterns that prevent us from using FEC packets for recovery). To address such loss patterns, we run our mirroring solution over TCP, which in turn runs over Maelstrom: if Maelstrom fails to recover a lost packet, the end-to-end TCP protocol will recover it from the sender.

3.3 Discussion

The key metric for any disaster-tolerant remote mirroring technology is the distance by which datacenters can be separated. Today, a disturbing number of New York City banks maintain backups in New Jersey or Brooklyn, because they simply cannot tolerate higher latencies.

The underlying problem is that these systems typically operate over TCP/IP. Obviously, the operators tune the system to match the properties of the network. For example, TCP can be configured to use massive sender buffers and unusually large segments; also, an application can be modified to employ multiple side-by-side streams (e.g. GridFTP). Yet even with such steps, the protocol remains purely *reactive*—recovery packets are sent only in response to actual indications of failure, in the form of negative acknowledgments (i.e. fast retransmit) or timeouts keyed to the round-trip-time (RTT). Consequently, their recovery time is tightly linked to the distance between communicating end-points. TCP/IP, for example, requires a minimum of around 1.5 RTTs to recover lost data, which translates into substantial fractions of a second if the mirrors are on different continents. No matter how large we make the TCP buffers, the remote data stream will experience an RTT hiccup each time loss occurs: to deliver data in order, the receiver must await the missing data before subsequent packets can be delivered.

Network-sync evades this RTT issue, but does not protect the application against every possible rolling disaster scenario. Packets can still be queued in the local-area when disaster strikes. Further, the network can partitioned in the split second(s) before a primary site fails. Neither proactive redundancy or network-level callbacks will prevent loss in these cases. Accordingly, we envision that applications will need a mixture of remote-sync and network-sync, with the former reserved for particularly sensitive scenarios, and the latter used in most cases.

Another issue is failover and recovery. Since the network-sync option enhances remote mirroring protocols, we assume that a complete remote mirroring protocol will itself handle failover and recovery directly [19, 22, 20]. As a result, in this work, we focus on evaluating the fault tolerant capabilities of a network-sync option and do not discuss failover and recovery protocols.

4 Mirroring Consistency via SMFS

We will say that a mirror image is *inconsistent* if out of order updates are applied to the mirror, or the application updates a group of files, and a period ensues during which some of the mirrored copies reflect the updates but others are stale. Inconsistency is a well-known problem when using networks to access file systems, and the issue can be exacerbated when mirroring. For example,

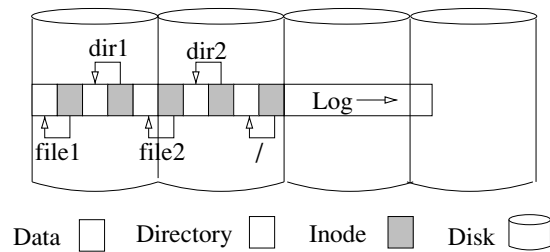


Figure 4: Format of a log after writing a file system with two sub directories `/dir1/file1` and `/dir2/file2`.

suppose that one were to mirror an NFS server, using the standard but unreliable UDP transport protocol. Primary and remote file systems can easily become inconsistent, since UDP packets can be reordered on the wire, particularly if a packet is dropped and the NFS protocol is forced to resend it. Even if a reliable transport protocol is used, in cases where the file system is spread over multiple storage servers, or applications update groups of files, skew in update behavior between the different mirrored servers may be perceived as an inconsistency by applications.

To address this issue, SMFS implements a file system that preserves the order of operations in the structure of the file system itself, a distributed log-structured file system (distributed-LFS)², where a particular log is distributed over multiple disks. Similar to LFS [35, 27], it embeds a UNIX tree-structured file system into an append only log format (Figure 4). It breaks a particular log into multiple segments that each have a finite maximum size and are the units of storage allocation and cleaning.

Although log-structured file systems may be unpopular in general settings (due to worries about high cleaning costs if the file system fills up), a log structure turns out to be nearly ideal for file mirroring. First, it is well known that an append-only log-structure is optimized for write performance [27, 35]. Second, by combining data and order of operations into one structure — the log — identical structures can be managed naturally at remote locations. Finally, log operations can be pipelined, increasing system throughput. Of course, none of this eliminates worries about segment cleaning costs. Our assumption is that because SMFS would be used only for files that need to be mirrored, such as backups and checkpoints, it can be configured with ample capacity—far from the tipping point at which these overheads become problematic.

In Sections 4.1 and 4.2, we describe the storage systems architecture and API.

4.1 SMFS Architecture

The SMFS architecture is illustrated in Figure 5. It works as follows. Clients access file system data by communi-

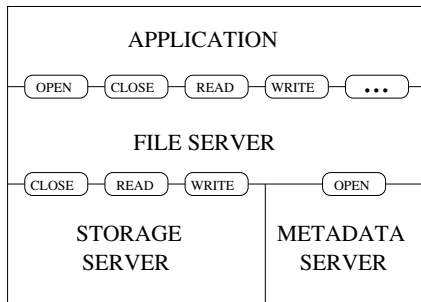


Figure 5: File System Architecture: Applications communicate with the *file server* through (possibly) a NFS interface. The file server in turn communicates with the *metadata server* through the `create()` function call. The metadata server allocates space for the newly created log on storage servers that it selects. The file server then interacts directly with the *storage server* for `append()`, `read()`, and `free()` operations.

cating with a file server (e.g. using the NFS protocol). File servers handle writes in a similar fashion to LFS. The log is updated by traversing a file system in a depth-first manner, first appending modified data blocks to the log, storing the log address in an inode, then appending the modified inode to the log, and so on [27, 29]. Reads are handled as in any conventional file system; starting with the root inode (stored in memory or a known location on disk) pointers are traversed to the desired file inode and data blocks. Although file servers provide a file system abstraction to clients, they are merely hosts in the storage system and stable storage resides with separate storage servers.

4.2 SMFS API

File servers interact with storage servers through a thin log interface—`create()`, `append()`, `read()`, and `free()`. `create()` communicates with a metadata server to allocate storage resources for a new log; it assigns responsibility for the new log to a storage server. After a log has been created, a file server uses the `append()` operation to add data to the log. The file server communicates directly with a log’s storage server to append data. The storage server assigns the order of each append—assigns the address in the log to a particular append—and atomically commits the operation. SMFS maintains group-mirroring consistency, in which a single `append()` can contain updates to many different files where the group of updates will all be reflected by the storage system atomically, either all or none. `read()` returns the data associated with a log address. Finally, `free()` takes a log address and marks the address for later cleaning. In particular, after a block has been modified or file removed, the file system calls `free()` on all blocks that are no longer referenced. The

`create()`, `append()`, and `free()` operations are mirrored between the primary site and remote mirror.

5 Evaluation

In this section, we evaluate the network-sync remote mirroring option, running our SMFS prototype on Emulab [40] and the Cornell National Lambda Rail (NLR) Rings testbed [1].

5.1 Experimental Environment

The implementation of SMFS that we worked was implemented as a user-mode application coded in Java. SMFS borrows heavily from our earlier file system, Antiquity [39]; however, the log address was modified to be a segment identifier and offset into the segment. A hash of the block can optionally be computed, but it is used as a checksum instead of as part of the block address in the log. We focus our evaluation on the `append()` operation since that is by far the dominant operation mirrored between two sites.

SMFS uses the Maelstrom network appliance [10] as the implementation of the network-sync option. Maelstrom can run as a user-mode module, but for the experiments reported here, it was dropped into the operating system, where it runs as a Linux 2.6.20 kernel module with hooks into the kernel packet filter [2]. Packets destined for the opposite site are routed through a pair of Maelstrom appliances located at each site. More importantly, situating a network appliance at the egress and ingress router for each site creates a virtual link between the two sites, which presents many opportunities for increasing mirroring reliability and performance.

The Maelstrom egress router captures packets, which it processes to create redundant packets. The original IP packets are forwarded unaltered; the redundant packets are then sent to the ingress router using a UDP channel. The ingress router captures and stores a window consisting of the last K IP packets that it has seen. Upon receiving a redundant packet it checks it against the last K IP packets. If there is an opportunity to recover any lost IP packet it does so, and forwards the newly recovered IP packet through a raw socket to the intended destination. Note that each appliance works in both egress and ingress mode since we handle duplex traffic.

To implement network-sync redundancy feedback, the Maelstrom kernel module tracks each TCP flow and sends an acknowledgment to the sender. Each acknowledgment includes a byte offset from the beginning of the stream up to the most recent byte that was included in an error correcting packet that was sent to the ingress router.

We used the TCP Reno congestion control algorithm to communicate between mirrored storage systems for all experiments. We experimented with other congestion control algorithms such as cubic; however, the results

were nearly identical since we were measuring packets lost after a primary site failure due to a disaster.

We tested the setup on Emulab [40]; our topology emulates two clusters of eight machines each, separated by a wide-area high capacity link with 50 to 200 ms RTT and 1 Gbps. Each machine has one 3.0 GHz Pentium 64-bit Xeon processor with 2.0 GB of memory and a 146 GB disks. Nodes are connected locally via a gigabit Ethernet switch. We apply load to these deployments using up to 64 testers located on the same cluster as the primary. A single tester is an individual application that has only one outstanding request at a time. Figure 3 shows the topology of our Emulab experimental setup (with the difference that we used eight nodes per cluster, and not four). Throughout all subsequent experiments, link loss is random, independent and identically distributed. See Balakrishnan et al [10] for an analysis with bursty link loss. Finally, all experiments show the average and standard deviation over five runs.

The overall SMFS prototype is fast enough to saturate a gigabit wide-area link, hence our decision to work with a user-mode Java implementation has little bearing on the experiments we now report: even if SMFS was implemented in the kernel in C, the network would still be the bottleneck.

5.2 Evaluation Metrics

We identify the following three metrics to evaluate the efficacy of SMFS:

- **Data Loss:** What happens in the event of a disaster at the primary? For varying loss rates on the wide-area link, how much does the mirror site diverge from the primary? We want our system to minimize this divergence.
- **Latency:** Latency can be used to measure both performance and reliability. Application-perceived latency measures (perceived) performance. Mirroring latency, on the other hand, measures reliability. In particular, the lower the latency, and the smaller the spread of its distribution, the better the fidelity of the mirror to the primary.
- **Throughput:** Throughput is a good measure of performance. The property we desire from our system is that throughput should degrade gracefully with increasing link loss and latency. Also, for mirroring solutions that use forward error correcting (FEC) codes, there is a fundamental tradeoff between data reliability and goodput (i.e. application level throughput); proactive redundancy via FEC increases tolerance to link loss and latency, but reduces the maximum goodput due to the overhead of FEC codes. We focus on goodput.

Layered Interleaving FEC Params[10]	r	8
	c	3
Network-sync Parameters	segment size	100 MB
	append size	512 kB
	block size	4 kB
Experiment Parameters	expt duration	3 mins

Table 2: Experimental Configuration Parameters

For effective comparison, we define the following five configurations; all configurations use TCP to communicate between each pair of storage servers.

- **Local-sync:** This is the canonical state-of-the-art solution. It is a semi-synchronous solution. As soon as the request has been applied to the local storage image and the local kernel buffers a request to send a message to the remote mirror, the local storage server responds to the application; it does not wait for feedback from remote mirror, or even for the packet to be placed on the wire.
- **Remote-sync:** This is the other end of the spectrum. It is a synchronous solution. The local storage server waits for a storage-level acknowledgment from the remote mirror before responding to the application.
- **Network-sync:** This is SMFS running with a network-sync option, implemented by Maelstrom in the manner outlined in Section 3 (e.g. with TCP over FEC). The network-sync layer provides feedback after proactively injecting redundancy into the network. SMFS responds to the application after receiving these redundancy notification.
- **Local-sync+FEC:** As a comparison point, this scheme is the local-sync mechanism, with Maelstrom running on the wide-area link, but without network-level callbacks to report when FEC packets are placed on the wire (i.e. storage servers are unaware of the proactive redundancy). The local server permits the application to resume execution as soon as data has been written to the local storage system.
- **Remote-sync+FEC:** As a second comparison point, this scheme is the remote-sync mechanism, again using Maelstrom on the wide-area link but without upcalls when FEC packets are sent. The local server waits for the remote storage system to acknowledge updates.

These five SMFS configurations are evaluated on each of the above metrics, and their comparative performance is presented. The Network-sync, Local-sync+FEC, and

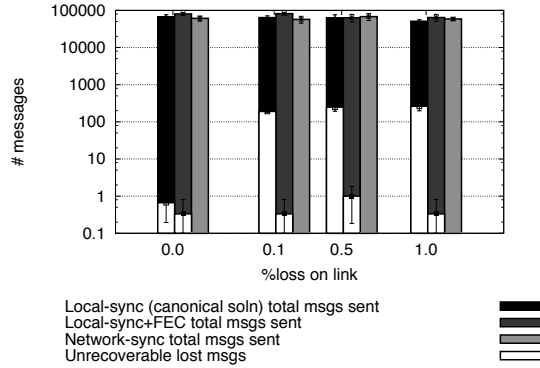


Figure 6: Data loss as a result of disaster and wide-area link failure, varying link loss (50ms one-way latency and FEC params $(r, c) = (8, 3)$).

Remote-sync+FEC configurations all use the Maelstrom layered interleaving forward error correction codes with parameters $(r, c) = (8, 3)$, which increases the tolerance to network transmission errors, but reduces the goodput by as much as 8/11 of the maximum throughput without any proactive redundancy. Table 2 lists the configuration parameters used in the experiments described below.

5.3 Reliability During Disaster

We measure reliability in two ways:

- In the event of a disaster at the primary site, how much data loss results?
- How much are the primary and mirror sites allowed to diverge?

These questions are highly related; we distinguish between them as follows: The maximum amount by which the primary and mirror sites can diverge is the extent of the bandwidth-delay product of the link between them; however, the amount of data lost in the event of failure depends on how much of this data has been acknowledged to the application. In other words, how often can we be caught in a lie? For instance, with a remote-sync solution (synchronous mirroring), though bandwidth-delay product – and hence primary-to-mirror divergence – may be high, data loss is zero. This, of course, is at severe cost to performance. With a local-sync solution (async- or semi-synchronous mirroring), on the other hand, data loss is equal to divergence. The following experiments show that the network-sync solution with SMFS achieves a desirable mean between these two extremes.

Disaster Test Figure 6 shows the amount of data loss in the event of a disaster for the local-sync, local-sync+FEC, and network-sync solutions; we do not test

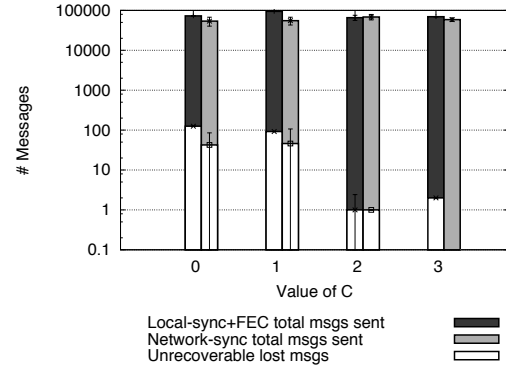


Figure 7: Data loss as a result of disaster and wide-area link failure, varying FEC param c (50ms one-way latency, 1% link loss).

the remote-sync and remote-sync+FEC solutions in this experiment since these solutions do not lose data.

The rolling disaster, failure of the wide-area link and crash of all primary site processes, occurred two minutes into the experiment. The wide-area link operated at 0% loss until immediately before the disaster occurred, when loss rate was increased for 0.5 seconds, thereafter the link was killed (See Section 2 for a description of rolling disasters). The x-axis shows the wide-area link loss rate immediately before the link is killed; link losses are random, independent and identically distributed. The y-axis shows both the total number of messages sent and total number of messages lost—lost messages were perceived as durable by the application but were not received by the remote mirror. Messages were of size 4 kB.

The total number of messages sent is similar for all configurations since the link loss rate was 0% for most of the experiment. However, local-sync lost a significant number of messages that had been reported to the application as durable under the policy discussed in Section 3.1. These unrecoverable messages were ones buffered in the kernel, but still in transit on the wide area link; when the sending datacenter crashed and the link (independently) dropped the original copy of the message, TCP recovery was unable to overcome the loss.

Local-sync+FEC lost packets as well: it lost packets still buffered in the kernel, but not packets that had already been transmitted — in the latter case, the proactive redundancy mechanism was adequate to overcome the loss. The best outcome is visible in the right-most histogram at 0.1%, 0.5%, and 1% link loss: here we see that although the network-sync solution experienced the same level of link-induced message loss, all the lost packets that had been reported as durable to the sender application were in fact recovered on the receiver side of the link. This supports the premise that a network-sync solution can tolerate disaster while minimizing loss.

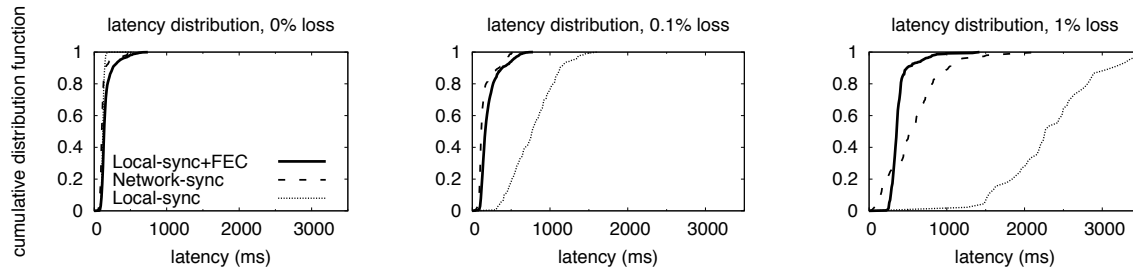


Figure 8: Latency distribution as a function of wide-area link loss (50ms one-way latency).

Combined with results from Section 5.4, we demonstrate that the network-sync solution actually achieves the best balance between reliability and performance.

Figure 7 quantifies the advantage of network-sync over local-sync+FEC. In this experiment, we run the same disaster scenario as above, but with 1% link loss during disaster and we vary the FEC parameter c (i.e. the number of recovery packets). At $c = 0$, there are no recovery packets for either local-sync+FEC or network-sync—if a data packet is lost during disaster, it cannot be recovered and TCP cannot deliver any subsequent data to the remote mirror process. Similarly, at $c = 1$, the number of lost packets is relatively high for both local-sync+FEC and network-sync since one recovery packet is not sufficient to mask 1% link loss. With $c > 1$, the number of recovery packets is often sufficient to mask loss on the wide-area link; however, local-sync+FEC loses data packets that did not transit outside the local-area before disaster, whereas with network-sync, primary storage servers respond to the client only after receiving a callback from the egress gateway. As a result, network-sync can potentially reduce data loss in a disaster.

Latency Figure 8 shows how latency is distributed across all requests for local-sync, local-sync+FEC, and network-sync solutions. Latency is the time between a local storage server sending a request and a remote storage server receiving the request. We see that these solutions show similar latency for zero link loss, but local-sync+FEC and network-sync show considerably better latency than local-sync for a lossy link. Furthermore, the latency spread of local-sync+FEC and network-sync solutions is considerably less than the spread of the local-sync solution — particularly as loss increases; proactive redundancy helps to reduce latency jitter on lossy links. Smaller variance in this latency distribution helps to ensure that updates submitted as a group will arrive at the remote site with minimum temporal skew, enabling the entire group to be written instead of not.

5.4 Performance

System Throughput Figure 9 compares the performance of the five different mirroring solutions. The x-axis shows loss probability on the wide-area link being

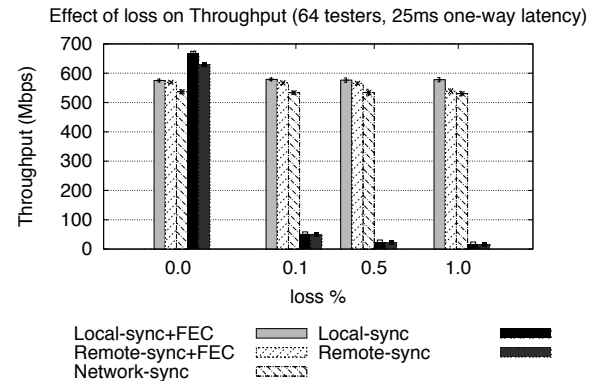


Figure 9: Effect of varying wide-area one-way link loss on Aggregate Throughput.

increased from 0% to 1%, while the y-axis shows the throughput achieved by each of these mirroring solutions. All mirroring solutions use 64 testers over eight storage servers.

At 0% loss we see that the local-sync and remote-sync solutions achieve the highest throughput because they do not use proactive redundancy, thus the goodput of the wide-area link is not reduced by the overhead of any forward error correcting packets. On the other hand, local-sync+FEC, remote-sync+FEC, and network-sync achieve lower throughput because the forward error correcting packets reduce the goodput in these cases. The forward error correction overhead is tunable; increasing FEC overhead often increases transmission reliability but reduces throughput. There is a slight degradation of performance for network-sync since SMFS waits for feedback from the egress router instead of responding immediately after the local kernel buffers the send request. Finally, the remote-sync and remote-sync+FEC achieve comparable performance to all the other configurations since there is no loss on the wide-area link and the storage servers can saturate the link with overlapping mirroring requests.

At higher loss rates, 0.1%, 0.5%, and 1%, we see that any solution that uses proactive redundancy (local-sync+FEC, remote-sync+FEC, and network-sync) achieves more than an order of magnitude higher

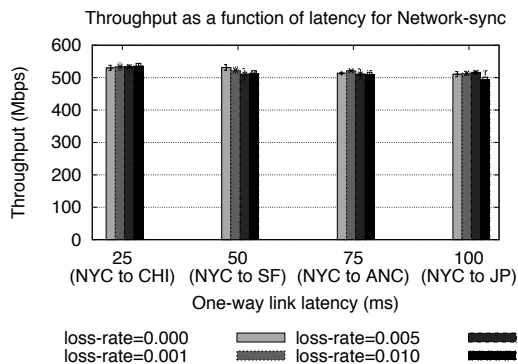


Figure 10: Effect of varying wide-area link *latency* on Aggregate Throughput.

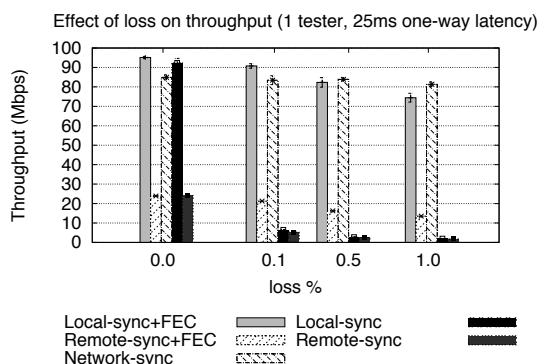


Figure 11: Effect of varying wide-area link *loss* on Per-Client Throughput.

throughput over any solution that does not. This illustrates the power of proactive redundancy, which makes it possible for these solutions to recover from lost packets at the remote mirror using locally-available data. Further, we observe that these proactive redundancy solutions perform comparably in both asynchronous and synchronous modes: in these experiments, the wide-area network is the bottleneck since overlapping operations can saturate the wide-area link.

Figure 10 shows the system throughput of the network-sync solution as the wide-area one-way link latency increases from 25 ms to 100 ms. It demonstrates that the network-sync solution (or any solution that uses proactive redundancy) can effectively mask latency and loss of a wide-area link.

Application Throughput The previous set of experiments studied system-level throughput, using a large number of testers. An interesting related study is presented here, of individual-application throughput in each SMFS configuration. Figure 11 shows the effect of increasing loss probability on the throughput of a application, with only one outstanding request at a time.

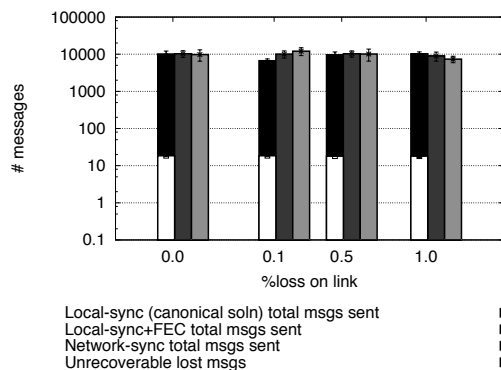


Figure 12: Data loss as a result of disaster and wide-area link failure (Cornell NLR-Rings, 37 ms one-way delay).

We see now that local-sync(+FEC) and network-sync solutions perform better than remote-sync(+FEC). The reason for this difference is that with asynchrony, network-sync can return an acknowledgment to the application as soon as a request is on the wide-area link, providing an opportunity to pipeline requests. This is in contrast to conventional asynchrony, where the application would receive an acknowledgment as soon as a request is *buffered*. The advantage with the former is that it provides performance gain without hurting reliability. The disadvantage is that pure buffering is a local system call operation, which can return to the application sooner and can achieve higher throughput as seen by the local-sync(+FEC) solutions. However, this increase in throughput is at a sacrifice of reliability; any buffered data may be lost in the event of a crash before it is sent (See Figure 6).

5.5 Cornell National Lambda Rail Rings

In addition to our emulated setup and results, we are beginning to physically study systems that operate on dedicated lambda networks that might be seen in cutting edge financial, military, and educational settings. To study these “personal” lambda networks, we have created a new testbed consisting of optical network paths of varying physical length that start and end at Cornell, the Cornell National Lambda Rail (NLR) Rings testbed.

The Cornell NLR-Rings testbed consists of three rings: a *short* ring that goes from Cornell to New York City and back, a *medium* ring that goes to Chicago down to Atlanta and back, and a *long* ring that goes to Seattle down to Los Angeles and back. The one-way latency is 7.9 ms, 37 ms, and 94 ms, for the short, medium, and long rings, respectively. The underlying optical networking technology is state-of-the-art: a 10 Gbps wide-area network running on dedicated fiber optics (separate from the public Internet) and created as a scientific research infrastructure by the NLR consortium [3]. Each ring

includes multiple segments of optical fiber, linked by routers and repeaters. More importantly, for the medium and long ring, each network packet traverses a unique path without going along the same segment. See NLR [3] for a map.

Though all rings in the testbed are capable of 10 Gbps end-to-end, we are only able to operate at hundreds of megabits per second at this time due to network construction. Nonetheless, we are able to study the effects of disaster on dedicated wide-area lambda networks and hope to be able to use increasingly more bandwidth in the future.

To study the effects of disaster in this wide-area testbed, we conduct the same disaster experiment described in Section 5.3. We induced loss on the wide-area link 0.5 second before the primary site fails via a router that we control. Later, when the primary site fails, the wide-area link and all processes are killed. Figure 12 shows data loss during this disaster for the medium path on the Cornell NLR-Rings testbed. The x-axis shows the loss induced on the wide-area link (link losses are random, independent and identically distributed) and the y-axis shows the number of messages sent and the number of unrecoverable messages. There are two interesting results illustrated. First, local-sync lost messages even when no loss was induced on the wide-area link. This may be because our wide-area testbed may drop packets, which prevents local-sync protocols from delivering to the mirroring application. Local-sync+FEC and network-sync, on the other hand, did not lose messages because both can mask wide-area link loss. Second, due to the relatively low bandwidth, packets were able to transit outside of the local-area, preventing loss from occurring in the local-area and enabling both local-sync+FEC and network-sync to mask wide-area link loss.

6 Related Work

6.1 Mirroring modes

Synchronous mirroring, like IBM's Peer-to-Peer Remote Copy (PPRC) [6] and EMC's Symmetrix Remote Data Facility (SRDF) [12] is a technique often used in disaster tolerance solutions. It guarantees that local copies of data are consistent with copies at a remote site, and also guarantees that the mirror sites are as up-to-date as possible. Naturally, the drawback is that of added I/O latency to every write operation; furthermore, long distance links make this technique prohibitively expensive.

An alternate solution is to use asynchronous remote mirroring [19, 24, 31]. For example, SnapMirror [31] provides asynchronous mirroring of file systems by periodically transferring self-consistent data snapshots from a source volume to a destination volume. Users are pro-

vided with a knob for setting the frequency of updates — if set to a high value, the mirror would be nearly current with the source, while setting to a low value reduces the network bandwidth consumption at the risk of increased data loss. Seneca [19] is a storage area network mirroring solution and similarly attempts to reduce the amount of traffic sent over the wide-area network.

SnapMirror works at the block level, using the WAFL [17] file system active block map to identify changed blocks and avoid sending deleted blocks. Moreover, since it operates at this level, it is able to optimize data reads and writes. The authors showed that for update intervals as short as one minute, data transfers were reduced by 30% to 80%.

Similar to SnapMirror, Seneca [19] is another asynchronous mirroring solution that attempts to reduce the traffic sent over the wide-area network, but also increases the risk of data loss. Seneca operates at the level of a storage area network (SAN) instead of the file system level.

Semi-synchronous mirroring is yet another mode of operation, closely related to both synchronous and asynchronous mirroring. In such a mode, writes are sent to both the local and the remote storage sites at the same time, the I/O operation returning when the local write is completed. However subsequent write I/O is delayed until the completion of the preceding remote write command. In [42] the authors show that by leveraging a log policy for the active remote write commands the system is able to allow a limited number of write I/O operations to proceed before waiting for acknowledgment from the remote site, thereby reducing the latency significantly.

6.2 Error correcting codes

Packet level forward error correcting (FEC) schemes typically transmit c repair packets for every r data packets, using coding schemes with which all data packets can be reconstructed if at least r out of $r + c$ data and repair packets are received [18]. In contrast, convolution codes work on bit or symbol streams of arbitrary length, and are most often decoded with the Viterbi algorithm [38]. Our work favors FEC: FEC schemes have the benefit of being highly tunable — trading off overhead and timeliness, and are very stable under stress — provided that the recovery does not result in high levels of traffic.

FEC techniques are increasingly popular. Recent applications include FEC for multicasting data to large groups [34], where FEC can be employed either by receivers [9] or senders [18, 28]. In general, fast, efficient encodings like Tornado codes [11] make sender-based FEC schemes very attractive in scenarios where dedicated senders distribute bulk data to a large number of receivers.

Likewise, FEC can be used when connections experience long transmission delays, in which case the use of

redundancy helps bound the delivery delays within some acceptable limits, even in the presence of errors [18, 33]. For example, deep space satellite communications [43] have been using error correcting codes for decades both for achieving maximal information transfer over a restricted bandwidth communication link and in the presence of data corruption.

SMFS is not the first system to propose exposing network state to higher level storage systems [32]. The difference, however, is that network-sync can be implemented with gateway routers under the control of site operators and does not require change to wide-area Internet routers.

6.3 Reliable Storage & Recovery

Recent studies have shown that failures plague storage and other components of large computing datacenters [36]. As a result, many systems replicate data to reduce risk of data loss [5, 14, 16, 25, 23, 37]. However, replication alone is not complete without recovery.

Recovery in the face of disaster has been a problem that has received a lot of attention [13, 21, 22]. In [20], for example, the authors propose a reactive way to solve the data recovery scheduling problem once the disaster has occurred. Potential recovery processes are first mapped onto recovery graphs — the recovery graphs capture alternative approaches for recovering workloads, precedence relationships, timing constraints, etc. The recovery scheduling problem is encoded as an optimization problem with the end goal of finding the schedule that minimizes some measure of penalty; several methods for finding optimal and near-optimal solutions are given.

Aguilera et. al. [4] explore the tradeoff between the ability to recover and the cost of recovery in enterprise storage systems. They propose a multi-tier file system called TierFS that employs a “recoverability log” used to increase the recoverability of lower tiers by using the highest tier.

Both LOCKSS [26] and Deep Store [44] address the problem of reliably preserving large volumes of data for virtually indefinite periods of time, dealing with threats like format obsolescence and “bit-rot.” LOCKSS consists of a set of low-cost, independent, persistent cooperating caches that use a voting scheme to detect and repair damaged content. Deep Store eliminates redundancy both within and across files; it distributes data for scalability and provides variable levels of replication based on the importance or the degree of dependency of each chunk of stored data.

Baker et. al. [8] consider the problem of recovery from failure of long-term storage of digital information. They propose a “reliability model” encompassing latent and correlated faults, and the detection time of such latent faults. They show that a simple combination of audit-

ing (to detect latent faults) as soon as possible, automatic recovery and independence of replicas yields the most benefit with respect to the cost of each technique.

7 Conclusion

The conundrum facing many disaster tolerance and recovery designs is the tradeoff between loss of performance and the potential loss of data. On the one hand, it may not be desirable to slow application response time until it is assured that data will not be lost in the event of disaster. On the other hand, the prospect of data loss can be catastrophic for many companies and organizations. Unfortunately, there is not much of a middle ground in the design space and designers must choose one or the other.

The network-sync remote mirroring option potentially offers an improvement, providing performance of enterprise-level semi-synchronous remote mirroring solutions while increasing their data reliability guarantees. Like native semi-synchronous protocols, network-sync protocols simultaneously send each update to the remote mirror as the primary handles the update locally. Rather than waiting for an acknowledgment from the remote mirror, it delays only until it receives feedback from an underlying communication layer, acknowledging that data and repair packets have been placed on the external wide-area network. This minimizes the loss of data in the event of disaster. Applications requiring strong remote-sync guarantees can still wait for a remote acknowledgment, but for most purposes, network-sync represents an appealing new option. Our experiments show that SMFS, a remote mirroring solution that uses the network-sync option, exhibits performance that is independent of link-latency, in marked contrast to most existing technologies.

Acknowledgments

We would like to thank our shepherd James Plank, the anonymous reviewers, and Robbert van Renesse for their comments that shaped the final version of this paper. Also, we would like to thank all who contributed to setting up the Cornell NLR-Rings testbed: Dan Freedman, Cornell Facilities Support Scott Yoest and Larry Parmelee, CIT-NCS networking engineering Eric Cronise and Dan Eckstrom, and NLR network engineering Greg Boles, Brent Sweeny, and Joe Lappa. Finally, we would like to thank Intel and Cisco for providing necessary routing and computing equipment, and NSF TRUST and AFRL Castor grant for funding support.

Notes

¹Egress and ingress routers operate as gateway routers between datacenter and wide-area networks, where egress routers send packets from local datacenter networks to the wide-area network and ingress

routers receive packets from the wide-area network and forward packets to local datacenter networks. Generally, egress routers also function as ingress routers and visa versa since they handle duplex traffic.

²A distributed log-structured file system can expose an NFS interface to hosts; however, it stores data in a distributed log-structured file system instead of a local UNIX file system (UFS).

References

- [1] Cornell national lambda rail (nlr) rings testbed. <http://www.cs.cornell.edu/~hweather/nlr>.
- [2] Firewalling, nat, and packet mangling for linux. <http://www.netfilter.org>. Last accessed Jan. 2009.
- [3] National lambda rail. <http://www.nlr.net>.
- [4] AGUILERA, M. K., KEETON, K., A, M., MUNISWAMY-REDDY, K., AND UYSAL, M. Improving recoverability in multi-tier storage systems. In *Proc. of DSN* (2007).
- [5] ANDERSON, T., DAHLIN, M., NEEFE, J., PATTERSON, D., ROSELLI, D., AND WANG, R. Serverless Network File Systems. In *Proc. of ACM SOSP* (Dec. 1995).
- [6] AZAGURY, A., FACTOR, M., AND MICKA, W. Advanced functions for storage subsystems: Supporting continuous availability. *An IBM SYSTEM Journal*, 2003.
- [7] BAIRAVASUNDARAM, L., GOODSON, G., PASUPATHY, S., AND SCHINDLER, J. An analysis of latent sector errors in disk drives. In *Proc. of ACM SIGMETRICS* (June 2007).
- [8] BAKER, M., SHAH, M., ROSENTHAL, D. S. H., ROUSSOPOULOS, M., MANIATIS, P., GIULI, T., AND BUNGAL, P. A fresh look at the reliability of long-term digital storage. *SIGOPS Oper. Syst. Rev.* 40, 4 (2006), 221–234.
- [9] BALAKRISHNAN, M., BIRMAN, K., PHANISHAYEE, A., AND PLEISCH, S. Ricochet: Lateral error correction for time-critical multicast. In *NSDI* (Apr. 2007).
- [10] BALAKRISHNAN, M., MARIAN, T., BIRMAN, K., WEATHERSPOON, H., AND VOLLSET, E. Maelstrom: Transparent error correction for lambda networks. In *NSDI* (2008).
- [11] BYERS, J. W., LUBY, M., MITZENMACHER, M., AND REGE, A. A digital fountain approach to reliable distribution of bulk data. *SIGCOMM Comput. Commun. Rev.* 28, 4 (1998), 56–67.
- [12] CORP, E. Symmetrix remote data facility. <http://www.emc.com/products/family/srdf-family.htm>. Last accessed Jan. 2009.
- [13] COUGIAS, D., HEIBERGER, E., AND KOOP, K. The backup book: disaster recovery from desktop to data center. Schaser-Vartan Books, Lecanto, FL, 2003.
- [14] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The google file system. In *Proc. of ACM SOSP* (Oct. 2003), pp. 29–43.
- [15] GUNAWI, H. S., PRABHAKARAN, V., KRISHNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Improving file system reliability with i/o shepherding. In *Proc. of ACM SOSP* (Oct. 2007).
- [16] HARTMAN, J. H., AND OUSTERHOUT, J. K. The zebra striped network file system. *ACM TOCS* (1995).
- [17] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In *Proceedings of the USENIX Technical Conference* (1994).
- [18] HUITEMA, H. The case for packet level FEC. In *Proc. of the IFIP Workshop on Protocols for High-Speed Networks* (Oct. 1996).
- [19] JI, M., VEITCH, A., AND WILKES, J. Seneca: Remote mirroring done write. In *Proc. of USENIX FAST* (June 2003).
- [20] KEETON, K., BEYER, D., BRAU, E., MERCHANT, A., SANTOS, C., AND ZHANG, A. On the road to recovery: Restoring data after disasters. In *Proc. of ACM EuroSys* (Apr. 2006).
- [21] KEETON, K., AND MERCHANT, A. A framework for evaluating storage system dependability. In *Proc. of DSN* (Washington, DC, USA, 2004), IEEE Computer Society, p. 877.
- [22] KEETON, K., SANTOS, C., BEYER, D., CHASE, J., AND WILKES, J. Designing for disasters. In *Proc. of USENIX FAST* (Berkeley, CA, USA, 2004), USENIX Association, pp. 59–62.
- [23] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *Proc. of ACM ASPLOS* (1996), pp. 84–92.
- [24] LEUNG, S. A., MACCORMICK, J., PERL, S. E., AND ZHANG, L. Myriad: Cost-effective disaster tolerance. In *Proc. of USENIX FAST* (Jan. 2002).
- [25] LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. Replication in the harp file system. In *Proc. of ACM SIGOPS* (1991).
- [26] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H., AND BAKER, M. The LOCKSS peer-to-peer digital preservation system. *ACM TOCS* (2005).
- [27] MATTHEWS, J., ROSELLI, D., COSTELLO, A., WANG, R., AND ANDERSON, T. Improving the performance of log-structured file systems with adaptive methods. In *Proc. of ACM SOSP* (1997).
- [28] NONNENMACHER, J., BIRSACK, E. W., AND TOWSLEY, D. Parity-based loss recovery for reliable multicast transmission. *IEEE/ACM Transactions on Networking* 6, 4 (1998), 349–361.
- [29] ORACLE. Berkeley db java edition architecture. An Oracle White Paper, Sept. 2006. <http://www.oracle.com/database/berkeley-db/jv/index.html>.
- [30] PARSONS, D. S., PERETTI, B., AND GROCHOW, J. M. Closing the gap: A research and development agenda to improve the resiliency of the banking and finance sector. In *U.S. Department of the Treasury Study* (Mar. 2005).
- [31] PATTERSON, H., MANLEY, S., FEDERWISCH, M., HITZ, D., KLEIMAN, S., AND OWARA, S. Snapmirror: File system based asynchronous mirroring for disaster recovery. In *Proc. of USENIX FAST* (Jan. 2002).
- [32] PLANK, J. S., BASSI, A., BECK, M., MOORE, T., SWANY, D. M., AND WOLSKI, R. Managing data storage in the network. *IEEE Internet Computing* 5, 5 (September/October 2001), 50–58.
- [33] RIZZO, L. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review* (1997).
- [34] RIZZO, L., AND VICISANO, L. RMDP: An fec-based reliable multicast protocol for wireless environments. In *ACM SIGCOMM Mobile Computer and Communication Review* (New York, NY, USA, 1998), vol. 2, ACM Press, pp. 22–31.
- [35] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems* 10, 1 (1992), 26–52.
- [36] SCHROEDER, B., AND GIBSON, G. A. Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? In *Proc. of USENIX FAST* (2007).
- [37] THEKKATH, C., MANN, T., AND LEE, E. Frangipani: A scalable distributed file system. In *Proc. of ACM SOSP* (1997).
- [38] VITERBI, A. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IT* 13 (1967), 260–269.
- [39] WEATHERSPOON, H., EATON, P., CHUN, B., AND KUBIATOWICZ, J. Antiquity: Exploiting a secure log for wide-area distributed storage. In *Proc. of ACM EuroSys* (2007).
- [40] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *Proc. of USENIX OSDI* (2002).
- [41] WITTY, R. J., AND SCOTT, D. Disaster recovery plans and systems are essential. *Gartner Research Gartner FirstTake: FT-14-5021* (September 12 2001).
- [42] YAN, R., SHU, J., AND CHAN WEN, D. An implementation of semi-synchronous remote mirroring system for sans. In *ACM Workshop of Grid and Cooperative Computing (GCC)* (2004).
- [43] YEUNG, R. W., AND ZHANG, Z. Distributed source coding for satellite communications. *IEEE Transactions on Information Theory* 45, 4 (May 1999), 1111–1120.
- [44] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep store: an archival storage system architecture. In *Proc. 21st International Conference on Data Engineering (ICDE)* (Apr. 2005).

Cumulus: Filesystem Backup to the Cloud

Michael Vrable, Stefan Savage, and Geoffrey M. Voelker

*Department of Computer Science and Engineering
University of California, San Diego*

Abstract

In this paper we describe Cumulus, a system for efficiently implementing filesystem backups over the Internet. Cumulus is specifically designed under a *thin cloud* assumption—that the remote datacenter storing the backups does not provide any special backup services, but only provides a least-common-denominator storage interface (i.e., get and put of complete files). Cumulus aggregates data from small files for remote storage, and uses LFS-inspired segment cleaning to maintain storage efficiency. Cumulus also efficiently represents incremental changes, including edits to large files. While Cumulus can use virtually any storage service, we show that its efficiency is comparable to integrated approaches.

1 Introduction

It has become increasingly popular to talk of “cloud computing” as the next infrastructure for hosting data and deploying software and services. Not surprisingly, there are a wide range of different architectures that fall under the umbrella of this vague-sounding term, ranging from highly integrated and focused (e.g., Software As A Service offerings such as Salesforce.com) to decomposed and abstract (e.g., utility computing such as Amazon’s EC2/S3). Towards the former end of the spectrum, complex logic is bundled together with abstract resources at a datacenter to provide a highly specific service—potentially offering greater performance and efficiency through integration, but also reducing flexibility and increasing the cost to switch providers. At the other end of the spectrum, datacenter-based infrastructure providers offer minimal interfaces to very abstract resources (e.g., “store file”), making portability and provider switching easy, but potentially incurring additional overheads from the lack of server-side application integration.

In this paper, we explore this *thin-cloud* vs. *thick-cloud* trade-off in the context of a very simple application: filesystem backup. Backup is a particularly attractive application for outsourcing to the cloud because it is relatively simple, the growth of disk capacity relative to tape capacity has created an efficiency and cost inflection point, and the cloud offers easy off-site storage, always a key concern for backup. For end users there are few backup solutions that are both trivial and reliable (especially against disasters such as fire or flood), and ubiq-

uitous broadband now provides sufficient bandwidth resources to offload the application. For small to mid-sized businesses, backup is rarely part of critical business processes and yet is sufficiently complex to “get right” that it can consume significant IT resources. Finally, larger enterprises benefit from backing up to the cloud to provide a business continuity hedge against site disasters.

However, to price cloud-based backup services attractively requires minimizing the capital costs of data center storage and the operational bandwidth costs of shipping the data there and back. To this end, most existing cloud-based backup services (e.g., Mozy, Carbonite, Symantec’s Protection Network) implement integrated solutions that include backup-specific software hosted on both the client and at the data center (usually using servers owned by the provider). In principle, this approach allows greater storage and bandwidth efficiency (server-side compression, cleaning, etc.) but also reduces portability—locking customers into a particular provider.

In this paper we explore the other end of the design space—the thin cloud. We describe a cloud-based backup system, called Cumulus, designed around a minimal interface (put, get, delete, list) that is trivially portable to virtually any on-line storage service. Thus, we assume that *any* application logic is implemented solely by the client. In designing and evaluating this system we make several contributions. First, we show through simulation that, through careful design, it is possible to build efficient network backup on top of a generic storage service—competitive with integrated backup solutions, in spite of having no specific backup support in the underlying storage service. Second, we build a working prototype of this system using Amazon’s Simple Storage Service (S3) and demonstrate its effectiveness on real end-user traces. Finally, we describe how such systems can be tuned *for cost* instead of for bandwidth or storage, both using the Amazon pricing model as well as for a range of storage to network cost ratios.

In the remainder of this paper, we first describe prior work in backup and network-based backup, followed by a design overview of Cumulus and an in-depth description of its implementation. We then provide both simulation and experimental results of Cumulus performance, overhead, and cost in trace-driven scenarios. We conclude with a discussion of the implications of our work

and how this research agenda might be further explored.

2 Related Work

Many traditional backup tools are designed to work well for tape backups. The `dump`, `cpio`, and `tar` [16] utilities are common on Unix systems and will write a full filesystem backup as a single stream of data to tape. These utilities may create a full backup of a filesystem, but also support *incremental* backups, which only contain files which have changed since a previous backup (either full or another incremental). Incremental backups are smaller and faster to create, but mostly useless without the backups on which they are based.

Organizations may establish backup policies specifying at what granularity backups are made, and how long they are kept. These policies might then be implemented in various ways. For tape backups, long-term backups may be full backups so they stand alone; short-term daily backups may be incrementals for space efficiency. Tools such as AMANDA [2] build on `dump` or `tar`, automating the process of scheduling full and incremental backups as well as collecting backups from a network of computers to write to tape as a group. Cumulus supports flexible policies for backup retention: an administrator does not have to select at the start how long to keep backups, but rather can delete any snapshot at any point.

The falling cost of disk relative to tape makes backup to disk more attractive, especially since the random access permitted by disks enables new backup approaches. Many recent backup tools, including Cumulus, take advantage of this trend. Two approaches for comparing these systems are by the storage representation on disk, and by the interface between the client and the storage—while the disk could be directly attached to the client, often (especially with a desire to store backups remotely) communication will be over a network.

Rsync [22] efficiently mirrors a filesystem across a network using a specialized network protocol to identify and transfer only those parts of files that have changed. Both the client and storage server must have `rsync` installed. Users typically want backups at multiple points in time, so `rsnapshot` [19] and other wrappers around `rsync` exist that will store multiple snapshots, each as a separate directory on the backup disk. Unmodified files are hard-linked between the different snapshots, so storage is space-efficient and snapshots are easy to delete.

The `rdiff-backup` [7] tool is similar to `rsnapshot`, but it changes the storage representation. The most recent snapshot is a mirror of the files, but the `rsync` algorithm creates compact deltas for reconstructing older versions—these reverse incrementals are more space efficient than full copies of files as in `rsnapshot`.

Another modification to the storage format at the server is to store snapshots in a content-addressable stor-

age system. Venti [17] uses hashes of block contents to address data blocks, rather than a block number on disk. Identical data between snapshots (or even within a snapshot) is automatically coalesced into a single copy on disk—giving the space benefits of incremental backups automatically. Data Domain [26] offers a similar but more recent and efficient product; in addition to performance improvements, it uses content-defined chunk boundaries so de-duplication can be performed even if data is offset by less than the block size.

A limitation of these tools is that backup data must be stored unencrypted at the server, so the server must be trusted. Box Backup [21] modifies the protocol and storage representation to allow the client to encrypt data before sending, while still supporting `rsync`-style efficient network transfers.

Most of the previous tools use a specialized protocol to communicate between the client and the storage server. An alternate approach is to target a more generic interface, such as a network file system or an FTP-like protocol. Amazon S3 [3] offers an HTTP-like interface to storage. The operations supported are similar enough between these different protocols—`get/put/delete` on files and `list` on directories—that a client can easily support multiple protocols. Cumulus tries to be network-friendly like `rsync`-based tools, while using only a generic storage interface.

Jungle Disk [13] can perform backups to Amazon S3. However, the design is quite different from that of Cumulus. Jungle Disk is first a network filesystem with Amazon S3 as the backing store. Jungle Disk can also be used for backups, keeping copies of old versions of files instead of deleting them. But since it is optimized for random access it is less efficient than Cumulus for pure backup—features like aggregation in Cumulus can improve compression, but are at odds with efficient random access.

Duplicity [8] aggregates files together before storage for better compression and to reduce per-file storage costs at the server. Incremental backups use space-efficient `rsync`-style deltas to represent changes. However, because each incremental backup depends on the previous, space cannot be reclaimed from old snapshots without another full backup, with its associated large upload cost. Cumulus was inspired by duplicity, but avoids this problem of long dependency chains of snapshots.

Brackup [9] has a design very similar to that of Cumulus. Both systems separate file data from metadata: each snapshot contains a separate copy of file metadata as of that snapshot, but file data is shared where possible. The split data/metadata design allows old snapshots to be easily deleted. Cumulus differs from Brackup primarily in that it places a greater emphasis on aggregating small files together for storage purposes, and adds a seg-

	Multiple snapshots	Simple server	Incremental forever	Sub-file delta storage	Encryption
rsync			✓	N/A	
rsnapshot	✓		✓		
rdiff-backup	✓		✓	✓	
Box Backup	✓		✓	✓	✓
Jungle Disk	✓	✓	✓		✓
duplicity	✓	✓		✓	✓
Brackup	✓	✓	✓		✓
Cumulus	✓	✓	✓	✓	✓

Multiple snapshots: Can store multiple versions of files at different points in time; *Simple server:* Can back up almost anywhere; does not require special software at the server; *Incremental forever:* Only initial backup must be a full backup; *Sub-file delta storage:* Efficiently represents small differences between files on storage; only relevant if storing multiple snapshots; *Encryption:* Data may be encrypted for privacy before sending to storage server.

Table 1: Comparison of features among selected tools that back up to networked storage.

ment cleaning mechanism to manage the inefficiency introduced. Additionally, Cumulus tries to efficiently represent small changes to all types of large files and can share metadata where unchanged; both changes reduce the cost of incremental backups.

Peer-to-peer systems may be used for storing backups. Pastiche [5] is one such system, and focuses on the problem of identifying and sharing data between different users. Pastiche uses content-based addressing for deduplication. But if sharing is not needed, Brackup and Cumulus could use peer-to-peer systems as well, simply treating it as another storage interface offering get and put operations.

While other interfaces to storage may be available—Antiquity [24] for example provides a log append operation—a get/put interface likely still works best since it is simpler and a single put is cheaper than multiple appends to write the same data.

Table 1 summarizes differences between some of the tools discussed above for backup to networked storage. In relation to existing systems, Cumulus is most similar to duplicity (without the need to occasionally re-upload a new full backup), and Brackup (with an improved scheme for incremental backups including rsync-style deltas, and improved reclamation of storage space).

3 Design

In this section we present the design of our approach for making backups to a thin cloud remote storage service.

3.1 Storage Server Interface

We assume only a very narrow interface between a client generating a backup and a server responsible for storing the backup. The interface consists of four operations:

Get: Given a pathname, retrieve the contents of a file from the server.

Put: Store a complete file on the server with the given pathname.

List: Get the names of files stored on the server.

Delete: Remove the given file from the server, reclaiming its space.

Note that all of these operations operate on entire files; we do not depend upon the ability to read or write arbitrary byte ranges within a file. Cumulus neither requires nor uses support for reading and setting file attributes such as permissions and timestamps. The interface is simple enough that it can be implemented on top of any number of protocols: FTP, SFTP, WebDAV, S3, or nearly any network file system.

Since the only way to modify a file in this narrow interface is to upload it again in full, we adopt a *write-once storage model*, in which a file is never modified after it is first stored, except to delete it to recover space. The write-once model provides convenient failure guarantees: since files are never modified in place, a failed backup run cannot corrupt old snapshots. At worst, it will leave a partially-written snapshot which can garbage-collected. Because Cumulus does not modify files in place, we can keep snapshots at multiple points in time simply by not deleting the files that make up old snapshots.

3.2 Storage Segments

When storing a snapshot, Cumulus will often group data from many smaller files together into larger units called *segments*. Segments become the unit of storage on the server, with each segment stored as a single file. Filesystems typically contain many small files (both our traces described later and others, such as [1], support this observation). Aggregation of data produces larger files for storage at the server, which can be beneficial to:

Avoid inefficiencies associated with many small files: Storage servers may dislike storing many small files for various reasons—higher metadata costs, wasted space from rounding up to block boundaries, and more seeks when reading. This preference may be expressed in the

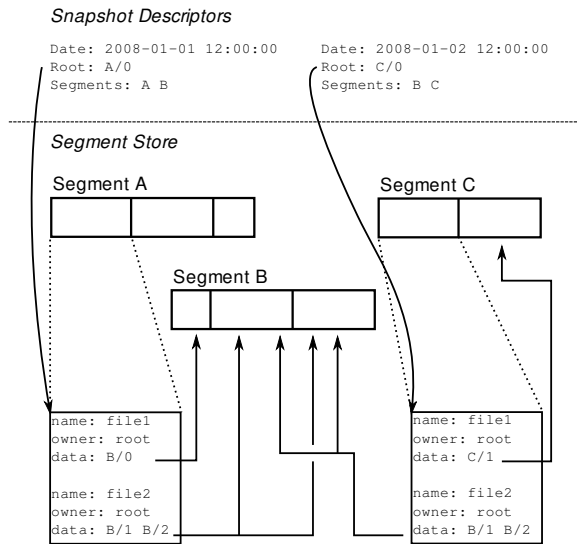


Figure 1: Simplified schematic of the basic format for storing snapshots on a storage server. Two snapshots are shown, taken on successive days. Each snapshot contains two files. `file1` changes between the two snapshots, but the data for `file2` is shared between the snapshots. For simplicity in this figure, segments are given letters as names instead of the 128-bit UUIDs used in practice.

cost model of the provider. Amazon S3, for example, has both a per-request and a per-byte cost when storing a file that encourages using files greater than 100 KB in size.

Avoid costs in network protocols: Small files result in relatively larger protocol overhead, and may be slower over higher-latency connections. Pipelining (if supported) or parallel connections may help, but larger segments make these less necessary. We study one instance of this effect in more detail in Section 5.4.5.

Take advantage of inter-file redundancy with segment compression: Compression can be more effective when small files are grouped together. We examine this effect in Section 5.4.2.

Provide additional privacy when encryption is used: Aggregation helps hide the size as well as contents of individual files.

Finally, as discussed in Sections 3.4 and 4.3, changes to small parts of larger files can be efficiently represented by effectively breaking those files into smaller pieces during backup. For the reasons listed above, re-aggregating this data becomes even more important when sub-file incremental backups are supported.

3.3 Snapshot Format

Figure 1 illustrates the basic format for backup snapshots. Cumulus snapshots logically consist of two parts: a *metadata log* which lists all the files backed up, and the

file data itself. Both metadata and data are broken apart into blocks, or *objects*, and these objects are then packed together into *segments*, compressed as a unit and optionally encrypted, and stored on the server. Each segment has a unique name—we use a randomly generated 128-bit UUID so that segment names can be assigned without central coordination. Objects are numbered sequentially within a segment.

Segments are internally structured as a TAR file, with each file in the archive corresponding to an object in the segment. Compression and encryption are provided by filtering the raw segment data through `gzip`, `bzip2`, `pgp`, or other similar external tools.

A snapshot can be decoded by traversing the tree (or, in the case of sharing, DAG) of objects. The root object in the tree is the start of the metadata log. The metadata log need not be stored as a flat object; it may contain pointers to objects containing other pieces of the metadata log. For example, if many files have not changed, then a single pointer to a portion of the metadata for an old snapshot may be written. The metadata objects eventually contain entries for individual files, with pointers to the file data as the leaves of the tree.

The metadata log entry for each individual file specifies properties such as modification time, ownership, and file permissions, and can be extended to include additional information if needed. It includes a cryptographic hash so that file integrity can be verified after a restore. Finally, it includes a list of pointers to objects containing the file data. Metadata is stored in a text, not binary, format to make it more transparent. Compression applied to the segments containing the metadata, however, makes the format space-efficient.

The one piece of data in each snapshot not stored in a segment is a *snapshot descriptor*, which includes a timestamp and a pointer to the root object.

Starting with the root object stored in the snapshot descriptor and traversing all pointers found, a list of all segments required by the snapshot can be constructed. Since segments may be shared between multiple snapshots, a garbage collection process deletes unreferenced segments when snapshots are removed. To simplify garbage-collection, each snapshot descriptor includes (though it is redundant) a summary of segments on which it depends.

Pointers within the metadata log include cryptographic hashes so that the integrity of all data can be validated starting from the snapshot descriptor, which can be digitally signed. Additionally, Cumulus writes a summary file with checksums for all segments so that it can quickly check snapshots for errors without a full restore.

3.4 Sub-File Incrementals

If only a small portion of a large file changes between snapshots, only the changed portion of the file should be stored. The design of the Cumulus format supports this. The contents of each file is specified as a list of objects, so new snapshots can continue to point to old objects when data is unchanged. Additionally, pointers to objects can include byte ranges to allow portions of old objects to be reused even if some data has changed. We discuss how our implementation identifies data that is unchanged in Section 4.3.

3.5 Segment Cleaning

When old snapshots are no longer needed, space is reclaimed by deleting the root snapshot descriptors for those snapshots, then garbage collecting unreachable segments. It may be, however, that some segments only contain a small fraction of useful data—the remainder of these segments, data from deleted snapshots, is now wasted space. This problem is similar to the problem of reclaiming space in the Log-Structured File System (LFS) [18].

There are two approaches that can be taken to segment cleaning given that multiple backup snapshots are involved. The first, *in-place cleaning*, is most like the cleaning in LFS. It identifies segments with wasted space and rewrites the segments to keep just the needed data.

This mode of operation has several disadvantages, however. It violates the write-once storage model, in that the data on which a snapshot depends is changed after the snapshot is written. It requires detailed book-keeping to determine precisely which data must be retained. Finally, it requires downloading and decrypting old segments—normal backups only require an encryption key, but cleaning needs the decryption key as well.

The alternative to in-place cleaning is to never modify segments in old snapshots. Instead, Cumulus avoids referring to data in inefficient old segments when creating a new snapshot, and writes new copies of that data if needed. This approach avoids the disadvantages listed earlier, but is less space-efficient. Dead space is not reclaimed until snapshots depending on the old segments are deleted. Additionally, until then data is stored redundantly since old and new snapshots refer to different copies of the same data.

We analyzed both approaches to cleaning in simulation. We found that the cost benefits of in-place cleaning were not large enough to outweigh its disadvantages, and so our Cumulus prototype does not clean in place.

The simplest policy for selecting segments to clean is to set a minimum segment utilization threshold, α , that triggers cleaning of a segment. We define utilization as the fraction of bytes within the segment which are ref-

erenced by a current snapshot. For example, $\alpha = 0.8$ will ensure that at least 80% of the bytes in segments are useful. Setting $\alpha = 0$ disables segment cleaning altogether. Cleaning thresholds closer to 1 will decrease storage overhead for a single snapshot, but this more aggressive cleaning requires transferring more data.

More complex policies are possible as well, such as a cost-benefit evaluation that favors repacking long-lived segments. Cleaning may be informed by snapshot retention policies: cleaning is more beneficial immediately before creating a long-term snapshot, and cleaning can also consider which other snapshots currently reference a segment. Finally, segment cleaning may reorganize data, such as by age, when segments are repacked.

Though not currently implemented, Cumulus could use heuristics to group data by expected lifetime when a backup is first written in an attempt to optimize segment data for later cleaning (as in systems such as WOLF [23]).

3.6 Restoring from Backup

Restoring data from previous backups may take several forms. A *complete restore* extracts all files as they were on a given date. A *partial restore* recovers one or a small number of files, as in recovering from an accidental deletion. As an enhancement to a partial restore, all available versions of a file or set of files can be listed.

Cumulus is primarily optimized for the first form of restore—recovering all files, such as in the event of the total loss of the original data. In this case, the restore process will look up the root snapshot descriptor at the date to restore, then download all segments referenced by that snapshot. Since segment cleaning seeks to avoid leaving much wasted space in the segments, the total amount of data downloaded should be only slightly larger than the size of the data to restore.

For partial restores, Cumulus downloads those segments that contain metadata for the snapshot to locate the files requested, then locates each of the segments containing file data. This approach might require fetching many segments—for example, if restoring a directory whose files were added incrementally over many days—but will usually be quick.

Cumulus is not optimized for tracking the history of individual files. The only way to determine the list of changes to a file or set of files is to download and process the metadata logs for all snapshots. However, a client could keep a database of this information to allow more efficient queries.

3.7 Limitations

Cumulus is not designed to replace all existing backup systems. As a result, there are situations in which other systems will do a better job.

The approach embodied by Cumulus is for the client making a backup to do most of the work, and leave the backup itself almost entirely opaque to the server. This approach makes Cumulus portable to nearly any type of storage server. However, a specialized backup server could provide features such as automatically repacking backup data when deleting old snapshots, eliminating the overhead of client-side segment cleaning.

Cumulus, as designed, does not offer coordination between multiple backup clients, and so does not offer features such as de-duplication between backups from different clients. While Cumulus could use convergent encryption [6] to allow de-duplication even when data is first encrypted at the client, several issues prevent us from doing so. Convergent encryption would not work well with the aggregation in Cumulus. Additionally, server-side de-duplication is vulnerable to dictionary attacks to determine what data clients are storing, and storage accounting for billing purposes is more difficult.

Finally, the design of Cumulus is predicated on the fact that backing up each file on the client to a separate file on the server may introduce too much overhead, and so Cumulus groups data together into segments. If it is known that the storage server and network protocol can efficiently deal with small files, however, then grouping data into segments adds unnecessary complexity and overhead. Other disk-to-disk backup programs may be a better match in this case.

4 Implementation

We discuss details of the implementation of the Cumulus prototype in this section. Our implementation is relatively compact: only slightly over 3200 lines of C++ source code (as measured by SLOccount [25]) implementing the core backup functionality, along with another roughly 1000 lines of Python for tasks such as restores, segment cleaning, and statistics gathering.

4.1 Local Client State

Each client stores on its local disk information about recent backups, primarily so that it can detect which files have changed and properly reuse data from previous snapshots. This information could be kept on the storage server. However, storing it locally reduces network bandwidth and improves access times. We do not need this information to recover data from a backup so its loss is not catastrophic, but this local state does enable various performance optimizations during backups.

The client's local state is divided into two parts: a local copy of the metadata log and an SQLite database [20] containing all other needed information.

Cumulus uses the local copy of the previous metadata log to quickly detect and skip over unchanged files based

on modification time. Cumulus also uses it to delta-encode the metadata log for new snapshots.

An SQLite database keeps a record of recent snapshots and all segments and objects stored in them. The table of objects includes an index by content hash to support data de-duplication. Enabling de-duplication leaves Cumulus vulnerable to corruption from a hash collision [11, 12], but, as with other systems, we judge the risk to be small. The hash algorithm (currently SHA-1) can be upgraded as weaknesses are found. In the event that client data must be recovered from backup, the content indices can be rebuilt from segment data as it is downloaded during the restore.

Note that the Cumulus backup format does not specify the format of this information stored locally. It is entirely possible to create a new and very different implementation which nonetheless produces backups conforming to the structure described in Section 3.3 and readable by our Cumulus prototype.

4.2 Segment Cleaning

The Cumulus backup program, written in C++, does not directly implement segment cleaning heuristics. Instead, a separate Cumulus utility program, implemented in Python, controls cleaning.

When writing a snapshot, Cumulus records in the local database a summary of all segments used by that snapshot and the fraction of the data in each segment that is actually referenced. The Cumulus utility program uses these summaries to identify segments which are poorly-utilized and marks the selected segments as “expired” in the local database. It also considers which snapshots refer to the segments, and how long those snapshots are likely to be kept, during cleaning. On subsequent backups, the Cumulus backup program re-uploads any data that is needed from expired segments. Since the database contains information about the age of all data blocks, segment data can be grouped by age when it is cleaned.

If local client state is lost, this age information will be lost. When the local client state is rebuilt all data will appear to have the same age, so cleaning may not be optimal, but can still be done.

4.3 Sub-File Incrementals

As discussed in Section 3.4, the Cumulus backup format supports efficiently encoding differences between file versions. Our implementation detects changes by dividing files into small *chunks* in a content-sensitive manner (using Rabin fingerprints) and identifying chunks that are common, as in the Low-Bandwidth File System [15].

When a file is first backed up, Cumulus divides it into blocks of about a megabyte in size which are stored individually in objects. In contrast, the chunks used for sub-file incrementals are quite a bit smaller: the target size is

4 KB (though variable, with a 2 KB minimum and 64 KB maximum). Before storing each megabyte block, Cumulus computes a set of chunk signatures: it divides the data block into non-overlapping chunks and computes a (20-byte SHA-1 signature, 2-byte length) tuple for each chunk. The list of chunk signatures for each object is stored in the local database. These signatures consume 22 bytes for every roughly 4 KB of original data, so the signatures are about 0.5% of the size of the data to back up.

Unlike LBFS, we do not create a global index of chunk hashes—to limit overhead, we do not attempt to find common data between different files. When a file changes, we limit the search for unmodified data to the chunks in the previous version of the file. Cumulus computes chunk signatures for the new file data, and matches with old chunks are written as a reference to the old data. New chunks are written out to a new object. However, Cumulus could be extended to perform global data deduplication while maintaining backup format compatibility.

4.4 Segment Filtering and Storage

The core Cumulus backup implementation is only capable of writing segments as uncompressed TAR files to local disk. Additional functionality is implemented by calling out to external scripts.

When performing a backup, all segment data may be filtered through a specified command before writing it. Specifying a program such as `gzip` can provide compression, or `gpg` can provide encryption.

Similarly, network protocols are implemented by calling out to external scripts. Cumulus first writes segments to a temporary directory, then calls an upload script to transfer them in the background while the main backup process continues. Slow uploads will eventually throttle the backup process so that the required temporary storage space is bounded. Upload scripts may be quite simple; a script for uploading to Amazon S3 is merely 12 lines long in Python using the `boto` [4] library.

4.5 Snapshot Restores

The Cumulus utility tool implements complete restore functionality. This tool can automatically decompress and extract objects from segments, and can efficiently extract just a subset of files from a snapshot.

To reduce disk space requirements, the restore tool downloads segments as needed instead of all at once at the start, and can delete downloaded segments as it goes along. The restore tool downloads the snapshot descriptor first, followed by the metadata. The backup tool segregates data and metadata into separate segments, so this phase does not download any file data. Then, file contents are restored—based on the metadata, as each

segment is downloaded data from that segment is restored. For partial restores, only the necessary segments are downloaded.

Currently, in the restore tool it is possible that a segment may be downloaded multiple times if blocks for some files are spread across many segments. However, this situation is merely an implementation issue and can be fixed by restoring data for these files non-sequentially as it is downloaded.

Finally, Cumulus includes a FUSE [10] interface that allows a collection of backup snapshots to be mounted as a virtual filesystem on Linux, thereby providing random access with standard filesystem tools. This interface relies on the fact that file metadata is stored in sorted order by filename, so a binary search can quickly locate any specified file within the metadata log.

5 Evaluation

We use both trace-based simulation and a prototype implementation to evaluate the use of thin cloud services for remote backup. Our goal is to answer three high-level sets of questions:

- What is the penalty of using a thin cloud service with a very simple storage interface compared to a more sophisticated service?
- What are the monetary costs for using remote backup for two typical usage scenarios? How should remote backup strategies adapt to minimize monetary costs as the ratio of network and storage prices varies?
- How does our prototype implementation compare with other backup systems? What are the additional benefits (e.g., compression, sub-file incrementals) and overheads (e.g., metadata) of an implementation not captured in simulation? What is the performance of using an online service like Amazon S3 for backup?

The following evaluation sections answer these questions, beginning with a description of the trace workloads we use as inputs to the experiments.

5.1 Trace Workloads

We use two traces as workloads to drive our evaluations. A **fileservers** trace tracks all files stored on our research group fileserver, and models the use of a cloud service for remote backup in an enterprise setting. A **user** trace is taken from the Cumulus backups of the home directory of one of the author's personal computers, and models the use of remote backup in a home setting. The traces contain a daily record of the metadata of all files in each setting, including a hash of the file contents. The user

	Fileserver	User
Duration (days)	157	223
Entries	26673083	122007
Files	24344167	116426
File Sizes		
Median	0.996 KB	4.4 KB
Average	153 KB	21.4 KB
Maximum	54.1 GB	169 MB
Total	3.47 TB	2.37 GB
Update Rates		
New data/day	9.50 GB	10.3 MB
Changed data/day	805 MB	29.9 MB
Total data/day	10.3 GB	40.2 MB

Table 2: Key statistics of the two traces used in the evaluations. File counts and sizes are for the last day in the trace. “Entries” is files plus directories, symlinks, etc.

trace further includes complete backups of all file data, and enables evaluation of the effects of compression and sub-file incrementals. Table 2 summarizes the key statistics of each trace.

5.2 Remote Backup to a Thin Cloud

First we explore the overhead of using remote backup to a thin cloud service that has only a simple storage interface. We compare this thin service model to an “optimal” model representing more sophisticated backup systems.

We use simulation for these experiments, and start by describing our simulator. We then define our optimal baseline model and evaluate the overhead of using a simple interface relative to a more sophisticated system.

5.2.1 Cumulus Simulator

The Cumulus simulator models the process of backing up collections of files to a remote backup service. It uses traces of daily records of file metadata to perform backups by determining which files have changed, aggregating changed file data into segments for storage on a remote service, and cleaning expired data as described in Section 3. We use a simulator, rather than our prototype, because a parameter sweep of the space of cleaning parameters on datasets as large as our traces is not feasible in a reasonable amount of time.

The simulator tracks three overheads associated with performing backups. It tracks storage overhead, or the total number of bytes to store a set of snapshots computed as the sum of the size of each segment needed. Storage overhead includes both actual file data as well as wasted space within segments. It tracks network overhead, the total data that must be transferred over the network to accomplish a backup. On graphs, we show this overhead as a cumulative value: the total data transferred from the beginning of the simulation until the given day.

Since remote backup services have per-file charges, the simulator also tracks segment overhead as the number of segments created during the process of making backups.

The simulator also models two snapshot scenarios. In the *single snapshot* scenario, the simulator maintains only one snapshot remotely and it deletes all previous snapshots. In the *multiple snapshot* scenario, the simulator retains snapshots according to a pre-determined backup schedule. In our experiments, we keep the most recent seven daily snapshots, with additional weekly snapshots retained going back farther in time so that a total of 12 snapshots are kept. This schedule emulates the backup policy an enterprise might employ.

The simulator makes some simplifying assumptions that we explore later when evaluating our implementation. The simulator detects changes to files in the traces using a per-file hash. Thus, the simulator cannot detect changes to only a portion of a file, and assumes that the entire file is changed. The simulator also does not model compression or metadata. We account for sub-file changes, compression, and metadata overhead when evaluating the prototype in Section 5.4.

5.2.2 Optimal Baseline

A simple storage interface for remote backup can incur an overhead penalty relative to more sophisticated approaches. To quantify the overhead of this approach, we use an idealized *optimal backup* as a basis of comparison.

For our simulations, the optimal backup is one in which no more data is stored or transferred over the network than is needed. Since simulation is done at a file granularity, the optimal backup will transfer the entire contents of a file if any part changes. Optimal backup will, however, perform data de-duplication at a file level, storing only one copy if multiple files have the same hash value. In the optimal backup, no space is lost to fragmentation when deleting old snapshots. Cumulus could achieve this optimal performance in this simulation by storing each file in a separate segment—that is, to never bundle files together into larger segments. As discussed in Section 3.2 and as our simulation results show, though, there are good reasons to use segments with sizes larger than the average file.

As an example of these costs and how we measure them, Figure 2(a) shows the optimal storage and upload overheads for daily backups of the 223 days of the user trace. In this simulation, only a single snapshot is retained each day. Storage grows slowly in proportion to the amount of data in a snapshot, and the cumulative network transfer grows linearly over time.

Figure 2(b) shows the results of two simulations of Cumulus backing up the same data. The graph shows the overheads relative to optimal backup; a backup as good as optimal would have 0% relative overhead. These re-

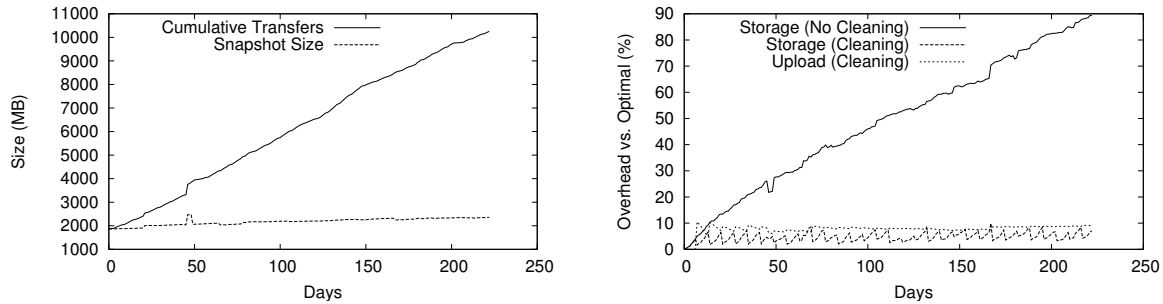


Figure 2: (a) Storage and network overhead for an optimal backup of the files from the user trace. (b) Overheads with and without cleaning; segments are cleaned at 60% utilization. Only storage overheads are shown for the no-cleaning case since there is no network transfer overhead without cleaning.

sults clearly demonstrate the need for cleaning when using a simple storage interface for backup. When segments are not cleaned (only deleting segments that by chance happen to be entirely no longer needed), wasted storage space grows quickly with time—by the end of the simulation at day 223, the size of a snapshot is nearly double the required size. In contrast, when segments are marked for cleaning at the 60% utilization threshold, storage overhead quickly stabilizes below 10%. The overhead in extra network transfers is similarly modest.

5.2.3 Cleaning Policies

Cleaning is clearly necessary for efficient backup, but it is also parameterized by two metrics: the size of the segments used for aggregation, transfer, and storage (Section 3.2), and the threshold at which segments will be cleaned (Section 3.5). In our next set of experiments, we explore the parameter space to quantify the impact of these two metrics on backup performance.

Figures 3 and 4 show the simulated overheads of backup with Cumulus using the fileserver and user traces, respectively. The figures show both relative overheads to optimal backup (left y -axis) as well as the absolute overheads (right y -axis). We use the backup policy of multiple daily and weekly snapshots as described in Section 5.2.1. The figures show cleaning overhead for a range of cleaning thresholds and segment sizes. Each figure has three graphs corresponding to the three overheads of remote backup to an online service. *Average daily storage* shows the average storage requirements per day over the duration of the simulation; this value is the total storage needed for tracking multiple backup snapshots, not just the size of a single snapshot. Similarly, *average daily upload* is the average of the data transferred each day of the simulation, excluding the first; we exclude the first day since any backup approach must transfer the entire initial filesystem. Finally, *average segments per day* tracks the number of new segments uploaded each day to account for per-file upload and storage costs.

Storage and upload overheads improve with decreasing segment size, but at small segment sizes (< 1 MB) backups require very large numbers of segments and limit the benefits of aggregating file data (Section 3.2). As expected, increasing the cleaning threshold increases the network upload overhead. Storage overhead with multiple snapshots, however, has an optimum cleaning threshold value. Increasing the threshold initially decreases storage overhead, but high thresholds increase it again; we explore this behavior further below.

Both the fileserver and user workloads exhibit similar sensitivities to cleaning thresholds and segment sizes. The user workload has higher overheads relative to optimal due to smaller average files and more churn in the file data, but overall the overhead penalties remain low.

Figures 3(a) and 4(a) show that there is a cleaning threshold that minimizes storage overheads. Increasing the cleaning threshold intuitively reduces storage overhead relative to optimal since the more aggressive cleaning at higher thresholds will delete wasted space in segments and thereby reduce storage requirements.

Figure 5 explains why storage overhead increases again at higher cleaning thresholds. It shows three curves, the 16 MB segment size curve from Figure 3(a) and two curves that decompose the storage overhead into individual components (Section 3.5). One is overhead due to duplicate copies of data stored over time in the cleaning process; cleaning at lower thresholds reduces this component. The other is due to wasted space in segments which have not been cleaned; cleaning at higher thresholds reduces this component. A cleaning threshold near the middle, however, minimizes the sum of both of these overheads.

5.3 Paying for Remote Backup

The evaluation in the previous section measured the overhead of Cumulus in terms of storage, network, and segment resource usage. Remote backup as a service, however, comes at a price. In this section, we calculate

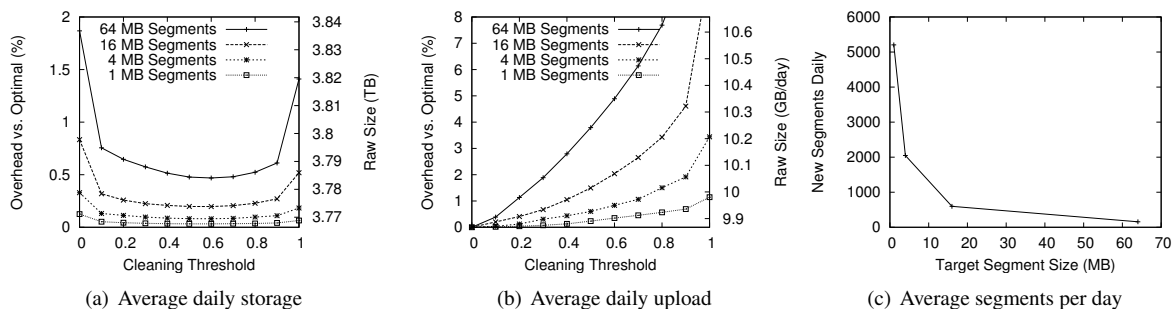


Figure 3: Overheads for backups in the fileserver trace.

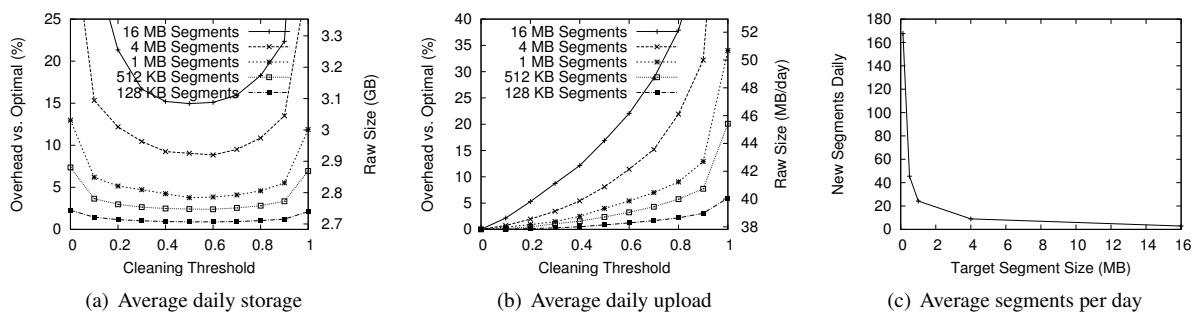


Figure 4: Overheads for backups in the user trace.

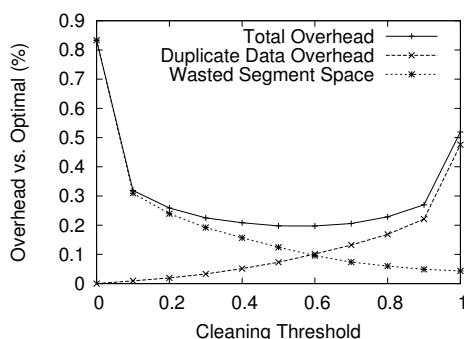


Figure 5: Detailed breakdown of storage overhead when using a 16 MB segment size for the fileserver workload.

monetary costs for our two workload models, evaluate cleaning threshold and segment size in terms of costs instead of resource usage, and explore how cleaning should adapt to minimize costs as the ratio of network and storage prices varies. While similar, there are differences between this problem and the typical evaluation of cleaning policies for a typical log-structured file system: instead of a fixed disk size and a goal to minimize I/O, we have no fixed limits but want to minimize monetary cost.

We use the prices for Amazon S3 as an initial point in the pricing space. As of January 2009, these prices are (in US dollars):

Fileserver	Amount	Cost
Initial upload	3563 GB	\$356.30
Upload	303 GB/month	\$30.30/month
Storage	3858 GB	\$578.70/month
User	Amount	Cost
Initial upload	1.82 GB	\$0.27
Upload	1.11 GB/month	\$0.11/month
Storage	2.68 GB	\$0.40/month

Table 3: Costs for backups in US dollars, if performed optimally, for the fileserver and user traces using current prices for Amazon S3.

Storage: \$0.15 per GB · month
Upload: \$0.10 per GB
Segment: \$0.01 per 1000 files uploaded

With this pricing model, the *segment* cost for uploading an empty file is equivalent to the *upload* cost for uploading approximately 100 KB of data, i.e., when uploading 100 KB files, half of the cost is for the bandwidth and half for the upload request itself. As the file size increases, the per-request component becomes an increasingly smaller part of the total cost.

Neglecting for the moment the segment upload costs, Table 3 shows the monthly storage and upload costs for each of the two traces. Storage costs dominate ongoing costs. They account for about 95% and 78% of the

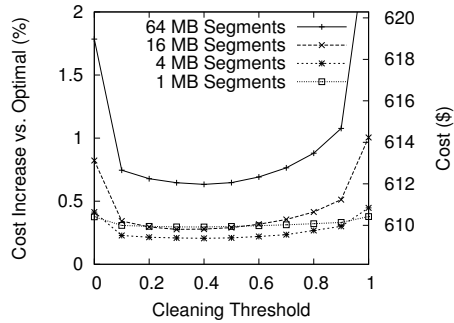


Figure 6: Costs in US dollars for backups in the fileserver assuming Amazon S3 prices. Costs for the user trace differ in absolute values but are qualitatively similar.

monthly costs for the fileserver and user traces, respectively. Thus, changes to the storage efficiency will have a more substantial effect on total cost than changes in bandwidth efficiency. We also note that the absolute costs for the home backup scenario are very low, indicating that Amazon’s pricing model is potentially quite reasonable for consumers: even for home users with an order of magnitude more data to backup than our user workload, yearly ongoing costs are roughly US\$50.

Whereas Figure 3 explored the parameter space of cleaning thresholds and segment sizes in terms of resource overhead, Figure 6 shows results in terms of overall cost for backing up the fileserver trace. These results show that using a simple storage interface for remote backup also incurs very low additional monetary cost than optimal backup, from 0.5–2% for the fileserver trace depending on the parameters, and as low as about 5% in the user trace.

When evaluated in terms of monetary costs, though, the choices of cleaning parameters change compared to the parameters in terms of resource usage. The cleaning threshold providing the minimum cost is smaller and less aggressive (threshold = 0.4) than in terms of resource usage (threshold = 0.6). However, since overhead is not overly sensitive to the cleaning threshold, Cumulus still provides good performance even if the cleaning threshold is not tuned optimally. Furthermore, in contrast to resource usage, decreasing segment size does not always decrease overall cost. At some point—in this case between 1–4 MB—decreasing segment size increases overall cost due to the per-file pricing. We do not evaluate segment sizes less than 1 MB for the fileserver trace since, by 1 MB, smaller segments are already a loss. The results for the user workload, although not shown, are qualitatively similar, with a segment size of 0.5 MB to 1 MB best.

The pricing model of Amazon S3 is only one point in the pricing space. As a final cost experiment, we ex-

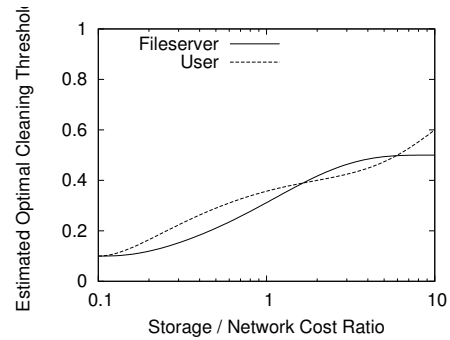


Figure 7: How the optimal threshold for cleaning changes as the relative cost of storage vs. network varies.

plore how cleaning should adapt to changes in the relative price of storage versus network. Figure 7 shows the optimal cleaning threshold for the fileserver and user workloads as a function of the ratio of storage to network cost. The storage to network ratio measures the relative cost of storing a gigabyte of data for a month and uploading a gigabyte of data. Amazon S3 has a ratio of 1.5. In general, as the cost of storage increases, it becomes advantageous to clean more aggressively (the optimal cleaning threshold increases). The ideal threshold stabilizes around 0.5–0.6 when storage is at least ten times more expensive than network upload, since cleaning too aggressively will tend to increase storage costs.

5.4 Prototype Evaluation

In our final set of experiments, we compare the overhead of the Cumulus prototype implementation with other backup systems. We also evaluate the sensitivity of compression on segment size, the overhead of metadata in the implementation, the performance of sub-file incrementals and restores, and the time it takes to upload data to a remote service like Amazon S3.

5.4.1 System Comparisons

First, we provide some results from running our Cumulus prototype and compare with two existing backup tools that also target Amazon S3: Jungle Disk and Brackup. We use the complete file contents included in the user trace to accurately measure the behavior of our full Cumulus prototype and other real backup systems. For each day in the first three months of the user trace, we extract a full snapshot of all files, then back up these files with each of the backup tools. We compute the average cost, per month, broken down into storage, upload bandwidth, and operation count (files created or modified).

We configured Cumulus to clean segments with less than 60% utilization on a weekly basis. We evaluate Brackup with two different settings. The first uses the `merge_files_under=1kB` option to only aggregate files if those files are under 1 KB in size

System	Storage	Upload	Operations
Jungle Disk	≈ 2 GB	1.26 GB	30000
	\$0.30	\$0.126	\$0.30
Brackup	1.340 GB	0.760 GB	9027
(default)	\$0.201	\$0.076	\$0.090
Brackup	1.353 GB	0.713 GB	1403
(aggregated)	\$0.203	\$0.071	\$0.014
Cumulus	1.264 GB	0.465 GB	419
	\$0.190	\$0.047	\$0.004

Table 4: Cost comparison for backups based on replaying actual file changes in the user trace over a three month period. Costs for Cumulus are lower than those shown in Table 3 since that evaluation ignored the possible benefits of compression and sub-file incrementals, which are captured here. Values are listed on a per-month basis.

(this setting is recommended). Since this setting still results in many small files (many of the small files are still larger than 1 KB), a “high aggregation” run sets `merge_files_under=16kB` to capture most of the small files and further reduce the operation count. Brackup includes the digest database in the files backed up, which serves a role similar to the database Cumulus stores locally. For fairness in the comparison, we subtract the size of the digest database from the sizes reported for Brackup.

Both Brackup and Cumulus use `gpg` to encrypt data in the test; `gpg` compresses the data with `gzip` prior to encryption. Encryption is enabled in Jungle Disk, but no compression is available.

In principle, we would expect backups with Jungle Disk to be near optimal in terms of storage and upload since no space is wasted due to aggregation. But, as a tradeoff, Jungle Disk will have a much higher operation count. In practice, Jungle Disk will also suffer from a lack of de-duplication, sub-file incrementals, and compression.

Table 4 compares the estimated backup costs for Cumulus with Jungle Disk and Brackup. Several key points stand out in the comparison:

- Storage and upload requirements for Jungle Disk are larger, owing primarily to the lack of compression.
- Except in the high aggregation case, both Brackup and Jungle Disk incur a large cost due to the many small files stored to S3. The per-file cost for uploads is larger than the per-byte cost, and for Jungle Disk significantly so.
- Brackup stores a complete copy of all file metadata with each snapshot, which in total accounts for 150–

200 MB/month of the upload cost. The cost in Cumulus is lower since Cumulus can re-use metadata.

Comparing storage requirements of Cumulus with the average size of a full backup with the venerable `tar` utility, both are within 1%: storage overhead in Cumulus is roughly balanced out by gains achieved from de-duplication. Using duplicity as a proxy for near-optimal incremental backups, in a test with two months from the user trace Cumulus uploads only about 8% more data than is needed. Without sub-file incrementals in Cumulus, the figure is closer to 33%.

The Cumulus prototype thus shows that a service with a simple storage interface can achieve low overhead, and that Cumulus can achieve a lower total cost than other existing backup tools targeting S3.

While perhaps none of the systems are yet optimized for speed, initial full backups in Brackup and Jungle Disk were both notably slow. In the tests, the initial Jungle Disk backup took over six hours, Brackup (to local disk, not S3) took slightly over two hours, and Cumulus (to S3) approximately 15 minutes. For comparison, simply archiving all files with `tar` to local disk took approximately 10 minutes.

For incremental backups, elapsed times for the tools were much more comparable. Jungle Disk averaged 248 seconds per run archiving to S3. Brackup averaged 115 seconds per run and Cumulus 167 seconds, but in these tests each were storing snapshots to local disk rather than to Amazon S3.

5.4.2 Segment Compression

Next we isolate the effectiveness of compression at reducing the size of the data to back up, particularly as a function of segment size and related settings. We used as a sample the full data contained in the first day of the user trace: the uncompressed size is 1916 MB, the compressed tar size is 1152 MB (factor of 1.66), and files individually compressed total 1219 MB (1.57×), 5.8% larger than whole-snapshot compression.

When aggregating data together into segments, we found that larger input segment sizes yielded better compression, up to about 300 KB when using `gzip` and 1–2 MB for `bzip2` where compression ratios leveled off.

5.4.3 Metadata

The Cumulus prototype stores metadata for each file in a backup snapshot in a text format, but after compression the format is still quite efficient. In the full tests on the user trace, the metadata for a full backup takes roughly 46 bytes per item backed up. Since most items include a 20-byte hash value which is unlikely to be compressible, the non-checksum components of the metadata average under 30 bytes per file.

	File A	File B
File size	4.860 MB	5.890 MB
Compressed size	1.547 MB	2.396 MB
Cumulus size	5.190 MB	3.081 MB
Size overhead	235%	29%
rdiff delta	1.421 MB	122 KB
Cumulus delta	1.527 MB	181 KB
Delta overhead	7%	48%

Table 5: Comparison of Cumulus sub-file incrementals with an idealized system based on rdiff, evaluated on two sample files from the user trace.

Metadata logs can be stored incrementally: new snapshots can reference the portions of old metadata logs that are not modified. In the full user trace replay, a full metadata log was written to a snapshot weekly. On days where only differences were written out, though, the average metadata log delta was under 2% of the size of a full metadata log. Overall, across all the snapshots taken, the data written out for file metadata was approximately 5% of the total size of the file data itself.

5.4.4 Sub-File Incrementals

To evaluate the support for sub-file incrementals in Cumulus, we make use of files extracted from the user trace that are frequently modified in place. We extract files from a 30-day period at the start of the trace. File A is a frequently-updated Bayesian spam filtering database, about 90% of which changes daily. File B records the state for a file-synchronization tool (unison), of which an average of 5% changes each day—however, unchanged content may still shift to different byte offsets within the file. While these samples do not capture all behavior, they do represent two distinct and notable classes of sub-file updates.

To provide a point of comparison, we use `rdiff` [14] to generate an `rsync`-style delta between consecutive file versions. Table 5 summarizes the results.

The *size overhead* measures the storage cost of sub-file incrementals in Cumulus. To reconstruct the latest version of a file, Cumulus might need to read data from many past versions, though cleaning will try to keep this bounded. This overhead compares the average size of a daily snapshot (“Cumulus size”) against the average compressed size of the file backed up. As file churn increases overhead tends to increase.

The *delta overhead* compares the data that must be uploaded daily by Cumulus (“Cumulus delta”) against the average size of patches generated by `rdiff` (“rdiff delta”). When only a small portion of the file changes each day (File B), `rdiff` is more efficient than Cumulus in representing the changes. However, sub-file incrementals are still a large win for Cumulus, as the size of the incre-

mentals is still much smaller than a full copy of the file. When large parts of the file change daily (File A), the efficiency of Cumulus approaches that of `rdiff`.

5.4.5 Upload Time

As a final experiment, we consider the time to upload to a remote storage service. Our Cumulus prototype is capable of uploading snapshot data directly to Amazon S3. To simplify matters, we evaluate upload time in isolation, rather than as part of a full backup, to provide a more controlled environment. Cumulus uses the `boto` [4] Python library to interface with S3.

As our measurements are from one experiment from a single computer (on a university campus network), they should not be taken as a good measure of the overall performance of S3. For large files—a megabyte or larger—uploads proceed at a maximum rate of about 800 KB/s. According to our results there is an overhead equivalent to a latency of roughly 100 ms per upload, and for small files this dominates the actual time for data transfer. It is thus advantageous to upload data in larger segments, as Cumulus does. More recent tests indicate that speeds may have improved.

The S3 protocol, based on HTTP, does not support pipelining multiple upload requests. Multiple uploads in parallel could reduce overhead somewhat. Still, it remains beneficial to perform uploads in larger units.

For perspective, assuming the maximum transfer rates above, ongoing backups for the fileserver and user workloads will take on average 3.75 hours and under a minute, respectively. Overheads from cleaning will increase these times, but since network overheads from cleaning are generally small, these upload times will not change by much. For these two workloads, backup times are very reasonable for daily snapshots.

5.4.6 Restore Time

To completely restore all data from one of the user snapshots takes approximately 11 minutes, comparable to but slightly faster than the time required for an initial full backup.

When restoring individual files from the user dataset, almost all time is spent extracting and parsing metadata—there is a fixed cost of approximately 24 seconds to parse the metadata to locate requested files. Extracting requested files is relatively quick, under a second for small files.

Both restore tests were done from local disk; restoring from S3 will be slower by the time needed to download the data.

6 Conclusions

It is fairly clear that the market for Internet-hosted backup service is growing. However, it remains unclear

what form of this service will dominate. On one hand, it is in the natural interest of service providers to package backup as an integrated service since that will both create a “stickier” relationship with the customer and allow higher fees to be charged as a result. On the other hand, given our results, the customer’s interest may be maximized via an open market for commodity storage services (such as S3), increasing competition due to the low barrier to switching providers, and thus driving down prices. Indeed, even today integrated backup providers charge between \$5 and \$10 per month per user while the S3 charges for backing up our test user using the Cumulus system was only \$0.24 per month. (For example, Symantec’s Protection Network charges \$9.99 per month for 10GB of storage and EMC’s MozyPro service costs \$3.95 + \$0.50/GB per month per desktop.)

Moreover, a thin-cloud approach to backup allows one to easily hedge against provider failures by backing up to multiple providers. This strategy may be particularly critical for guarding against business risk—a lesson that has been learned the hard way by customers whose hosting companies have gone out of business. Providing the same hedge using the integrated approach would require running multiple backup systems in parallel on each desktop or server, incurring redundant overheads (e.g., scanning, compression, etc.) that will only increase as disk capacities grow.

Finally, while this paper has focused on an admittedly simple application, we believe it identifies a key research agenda influencing the future of “cloud computing”: can one build a competitive product economy around a cloud of abstract commodity resources, or do underlying technical reasons ultimately favor an integrated service-oriented infrastructure?

7 Acknowledgments

The authors would like to thank Chris X. Edwards and Brian Kantor for assistance in collecting fileserver traces and other computing support. We would also like to thank our shepherd Niraj Tolia and the anonymous reviewers for their time and insightful comments regarding Cumulus and this paper. This work was supported in part by the National Science Foundation grant CNS-0433668 and the UCSD Center for Networked Systems. Vrable was further supported in part by a National Science Foundation Graduate Research Fellowship.

References

- [1] AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. A five-year study of file-system metadata. *ACM Trans. Storage* 3, 3 (2007), 9.
- [2] The Advanced Maryland Automatic Network Disk Archiver. <http://www.amanda.org/>.
- [3] AMAZON WEB SERVICES. Amazon Simple Storage Service. <http://aws.amazon.com/s3/>.
- [4] boto: Python interface to Amazon Web Services. <http://code.google.com/p/boto/>.
- [5] COX, L. P., MURRAY, C. D., AND NOBLE, B. D. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (2002), USENIX, pp. 285–298.
- [6] DOUCEUR, J. R., ADYA, A., BOLOSKY, W. J., SIMON, D., AND THEIMER, M. Reclaiming space from duplicate files in a serverless distributed file system. Technical Report MSR-TR-2002-30.
- [7] ESCOTO, B. rdiff-backup. <http://www.nongnu.org/rdiff-backup/>.
- [8] ESCOTO, B., AND LOAFMAN, K. Duplicity. <http://duplicity.nongnu.org/>.
- [9] FITZPATRICK, B. Brackup. <http://code.google.com/p/brackup/>, <http://brad.livejournal.com/tag/brackup>.
- [10] FUSE: Filesystem in userspace. <http://fuse.sourceforge.net/>.
- [11] HENSON, V. An analysis of compare-by-hash. *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems* (2003).
- [12] HENSON, V. The code monkey’s guide to cryptographic hashes for content-based addressing, Nov. 2007. <http://www.linuxworld.com/news/2007/111207-hash.html>.
- [13] Jungle disk. <http://www.jungledisk.com/>.
- [14] librsync. <http://librsync.sourceforge.net/>.
- [15] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (2001), ACM, pp. 174–187.
- [16] PRESTON, W. C. *Backup & Recovery*. O’Reilly, 2006.
- [17] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)* (2002), USENIX Association.
- [18] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (1992), 26–52.
- [19] rsnapshot. <http://www.rsnapshot.org/>.
- [20] Sqlite. <http://www.sqlite.org/>.
- [21] SUMMERS, B., AND WILSON, C. Box backup. <http://www.boxbackup.org/>.
- [22] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Feb. 1999.
- [23] WANG, J., AND HU, Y. WOLF—A novel reordering write buffer to boost the performance of log-structured file systems. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)* (2002), USENIX Association.
- [24] WEATHERSPOON, H., EATON, P., CHUN, B.-G., AND KUBIATOWICZ, J. Antiquity: Exploiting a secure log for wide-area distributed storage. In *EuroSys ’07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), ACM, pp. 371–384.
- [25] WHEELER, D. A. SLOccount. <http://www.dwheeler.com/sloccount/>.
- [26] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)* (2008), USENIX Association, pp. 269–282.

WorkOut: I/O Workload Outsourcing for Boosting RAID Reconstruction Performance

Suzhen Wu¹, Hong Jiang², Dan Feng^{1*}, Lei Tian¹², Bo Mao¹

¹Key Laboratory of Data Storage Systems, Ministry of Education of China

¹School of Computer Science & Technology, Huazhong University of Science & Technology

²Department of Computer Science & Engineering, University of Nebraska-Lincoln

*Corresponding author: dfeng@hust.edu.cn

{[suzhen66](mailto:suzhen66@gmail.com), [maobo.hust](mailto:maobo.hust@gmail.com)}@gmail.com, {[jiang](mailto:jiang@cse.unl.edu), [tian](mailto:tian@cse.unl.edu)}@cse.unl.edu, ltian@hust.edu.cn

Abstract

User I/O intensity can significantly impact the performance of on-line RAID reconstruction due to contention for the shared disk bandwidth. Based on this observation, this paper proposes a novel scheme, called *WorkOut* (I/O Workload Outsourcing), to significantly boost RAID reconstruction performance. WorkOut effectively outsources all write requests and popular read requests originally targeted at the degraded RAID set to a surrogate RAID set during reconstruction. Our lightweight prototype implementation of WorkOut and extensive trace-driven and benchmark-driven experiments demonstrate that, compared with existing reconstruction approaches, WorkOut significantly speeds up both the total reconstruction time and the average user response time. Importantly, WorkOut is orthogonal to and can be easily incorporated into any existing reconstruction algorithms. Furthermore, it can be extended to improving the performance of other background support RAID tasks, such as re-synchronization and disk scrubbing.

1 Introduction

As a fundamental technology for reliability and availability, RAID [30] has been widely deployed in modern storage systems. A RAID-structured storage system ensures that data will not be lost when disks fail. One of the key responsibilities of RAID is to recover the data that was on a failed disk, a process known as *RAID reconstruction*.

The performance of RAID reconstruction techniques depends on two factors. First, the time it takes to complete the reconstruction of a failed disk, since longer reconstruction times translate to a longer “window of vulnerability”, in which a second disk failure may cause persistent data loss. Second, the impact of the reconstruction process on the foreground workload, i.e., to what degree are user requests affected by the ongoing reconstruction.

Current approaches for RAID reconstruction fall into two different categories: [11, 12]: *off-line reconstruction*, when the RAID devotes all of its resources to perform-

ing reconstruction without serving any I/O requests from user applications, and *on-line reconstruction*, when the RAID continues to service user I/O requests during reconstruction.

Off-line reconstruction has the advantage that it's faster than on-line reconstruction, but it is not practical in environments with high availability requirements, as the entire RAID set needs to be taken off-line during reconstruction.

On the other hand, on-line reconstruction allows foreground traffic to continue during reconstruction, but takes longer to complete than off-line reconstruction as the reconstruction process competes with the foreground workload for I/O bandwidth. In our experiments we find that on-line reconstruction (with heavy user I/O workloads) can be as much as 70 times (70×) slower than off-line reconstruction (without user I/O workloads) (see Section 2.2). Moreover, while on-line reconstruction allows foreground workload to be served, the performance of the foreground workload might be significantly reduced. In our experiments, we see cases where the user response time increases by a factor of 3 (3×) during on-line reconstruction (see Section 2.2).

Improving the performance of on-line RAID reconstruction is becoming a growing concern in the light of recent technology trends: reconstruction times are expected to increase in the future, as the capacity of drives grows at a much higher rate than other performance parameters, such as bandwidth, seek time and rotational latency [10]. Moreover, with the ever growing number of drives in data centers, reconstruction might soon become the common mode of operation in large-scale systems rather than the exception [4, 9, 18, 32, 34].

A number of approaches have been proposed to improve the performance of RAID reconstruction, including for example optimizing the reconstruction workflow [12, 22], the reconstruction sequence [3, 41] or the data layout [14, 49]. We note that all these approaches focus on a single RAID set. In this paper we propose a new approach to improving reconstruction performance,

exploiting the fact that most data centers contain a large number of RAID sets.

Inspired by recent work on data migration [2, 24, 46] and write off loading [27], we propose *WorkOut*, a framework to significantly improve on-line reconstruction performance by *I/O Workload Outsourcing*. The main idea behind *WorkOut* is to temporarily redirect all write requests and popular read requests originally targeted at the degraded RAID set to a *surrogate RAID set*. The surrogate RAID set can be free space on another live RAID set or a set of spare disks.

The benefits of *WorkOut* are two-fold. *WorkOut* reduces the impact of reconstruction on foreground traffic because most user requests can be served from the surrogate RAID set and hence no longer compete with the reconstruction process for disk bandwidth. *WorkOut* also speeds up the reconstruction process, since more bandwidth on the degraded RAID set can be devoted to the reconstruction process.

In more detail, *WorkOut* has the following salient features:

- (1) *WorkOut* tackles one of the most important factors adversely affecting reconstruction performance, namely, *I/O intensity*, that, to the best of our knowledge, has not been adequately addressed by the previous studies [3, 41].
- (2) *WorkOut* has a distinctive advantage of improving both reconstruction time and user response time. It is a very effective reconstruction optimization scheme focusing on optimizing write-intensive workloads, a roadblock for many of the existing reconstruction approaches [41].
- (3) *WorkOut* is orthogonal and complementary to and can be easily incorporated into most existing RAID reconstruction approaches to further improve their performance.
- (4) In addition to boosting RAID reconstruction performance, *WorkOut* is very lightweight and can be easily extended to improve the performance of other background tasks, such as re-synchronization [7] and disk scrubbing [36], that are also becoming more frequent and lengthier for the same reasons that reconstruction is becoming more frequent and lengthier.

Extensive trace-driven and benchmark-driven experiments conducted on our lightweight prototype implementation of *WorkOut* show that *WorkOut* significantly outperforms the existing reconstruction approaches PR [22] and PRO [42] in both reconstruction time and user response time.

The rest of this paper is organized as follows. Background and motivation are presented in Section 2. We describe the design of *WorkOut* in Section 3. Performance

evaluations of *WorkOut* based on a prototype implementation are presented in Section 4. We analyze the reliability of *WorkOut* in Section 5 and present related work in Section 6. We point out directions for future research in Section 7 and summarize the main contributions of the paper in Section 8.

2 Background and Motivation

In this section, we provide some background and key observations that motivate our work and facilitate our presentation of *WorkOut* in later sections.

2.1 Disk failures in the real world

Recent studies of field data on partial or complete disk failures in large-scale storage systems indicate that disk failures happen at a significant rate [4, 9, 18, 32, 34]. Schroeder & Gibson [34] found that annual disk replacement rates in the real world exceed 1%, with 2%-4% on average and up to 13% in some systems, much higher than 0.88%, the annual failure rates (AFR) specified by the manufacturer's datasheet. Bairavasundaram *et al.* [4] observed that the probability of latent sector errors, which can lead to disk replacement, is 3.45% in their study. Those failure rates, combined with the continuously increasing number of drives in large-scale storage systems, raise concerns that in future storage systems, recovery mode might become the common mode of operation [9].

Another concern arises from recent studies showing a significant amount of correlation in drive failures, indicating that, after one disk fails, another disk failure will likely occur soon [4, 9, 18]. Gibson [9] also points out that the probability of a second disk failure in a RAID system during reconstruction increases along with the reconstruction time: approximately 0.5% for one hour, 1.0% for 3 hours and 1.4% for 6 hours.

All the above trends make fast recovery from disk failures an increasingly important factor in building storage systems.

2.2 Mutually adversary impact of reconstruction and user I/O requests

During on-line RAID reconstruction, reconstruction requests and user I/O requests compete for the bandwidth of the surviving disks and adversely affect each other. User I/O requests increase the reconstruction time while the reconstruction process increases the user response time.

Figure 1 shows the reconstruction times and user response times of a 5-disk RAID5 set with a stripe unit size of 64KB in three cases: (1) off-line reconstruction, (2) on-line reconstruction at the highest speed (when RAID favors the reconstruction process), and (3) on-line reconstruction at the lowest speed (when RAID favors user

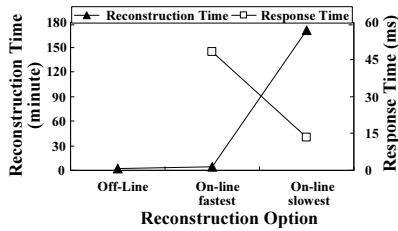


Figure 1: Reconstruction and its performance impact.

I/O requests). In this experiment we limit the capacity of each disk to 10GB. User I/O requests are generated by Iometer [17] with 20% sequential and 60%/40% read/write requests of 8KB each. As shown in Figure 1, the user response time increases significantly along with the reconstruction speed, 3 times ($3\times$) more than that in the normal mode. The on-line reconstruction process at the lowest speed takes 70 times ($70\times$) longer than its off-line counterpart.

How reconstruction is performed impacts both the reliability and availability of storage systems [11]. Storage system reliability is formally defined as MTDL (the mean time to data loss) and increases with decreasing MTTR (the mean time to repair). Ironically, decreasing the MTTR (i.e., speeding up reconstruction) by throttling foreground user requests can lead SLA (Service Level Agreement) violations, which in many environments are also perceived as reduced availability. Ideally, one would like to reduce both the reconstruction time and user response time in order to improve the reliability and availability of RAID-structured storage systems.

In Figure 2, we take a closer look at how user I/O intensity affects the performance of RAID reconstruction. The experimental setup is the same as that in Figure 1, except that we impose different I/O request intensities. Moreover, the RAID reconstruction process is set to yield to user I/O requests (i.e., RAID favors user I/O requests). From Figure 2, we see that both the reconstruction time and user response time increase with IOPS (*I/O Per Second*). When increasing the user IOPS from 9 to 200, reaches its maximum of 200, the reconstruction time increases by a factor of 20.9 and average user response time increases by a factor of 3.76.

From the above experiments and analysis, we believe that reducing the amount of user I/O requests directed to the degraded RAID set is an effective approach to simultaneously reducing the reconstruction time and alleviating the user performance degradation, thus improving both reliability and availability. However, naively redirecting *all* requests from the degraded RAID to a surrogate RAID might overload the surrogate RAID and runs the risk that a lot of work is wasted by redirecting requests that will never be accessed again. Our idea is therefore to exploit locality in the request stream and redirect only requests for *popular* data.

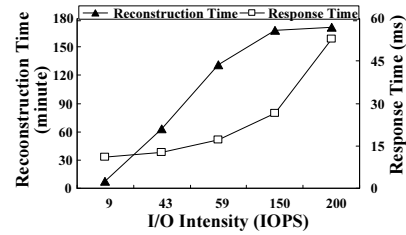


Figure 2: I/O intensity impact on reconstruction.

2.3 Workload locality

Previous studies indicate that access locality is one of the key web workload characteristics [1, 5, 6] and observe that 10% of files accessed on a web server approximately account for 90% of the requests and 90% of the bytes transferred [1]. Such studies also find that 20%-40% of the files are accessed only once for web workloads [6].

To exploit access locality, caches have been widely employed to improve storage system performance. Storage caches, while proven very effective in capturing workload locality, is so small in capacity compared with the typical storage device that it usually cannot capture all workload locality. Thus the locality underneath the storage cache can still be effectively mined and utilized [23, 41]. For example, based on the study on C-Miner [23] that mines block correlation below the storage cache, correlation-directed prefetching and data layout help reduce the user response time of the baseline case by 12-25%. By utilizing the workload locality at the block level, PRO [41] reduces the reconstruction time by up to 44.7% and the user response time by 3.6-23.9% simultaneously.

Based on these observations, WorkOut only redirects the popular read data to the surrogate RAID set to exploit access locality of read requests bound to the most popular data. *For simplicity, popular data in our design is defined as the data that has been read at least twice during reconstruction.* Different from read requests, write requests can be served by any persistent storage device. Thus WorkOut redirects all write requests to the surrogate RAID set.

3 WorkOut

In this section, we first outline the main principles guiding the design of WorkOut. Then we present an architectural overview of WorkOut, followed by a description of the WorkOut organization and algorithm. The design choice and data consistency issues of WorkOut are discussed at the end of this section.

3.1 Design principles

WorkOut focuses on outsourcing I/O workloads and aims to achieve reliability, availability, extendibility and flexibility, as follows.

Reliability. To reduce the window of vulnerability and thus improve the system reliability, the reconstruction time must be significantly reduced. Since user I/O intensity severely affects the reconstruction process, WorkOut aims to reduce the I/O intensity on the degraded RAID set by redirecting I/O requests away from the degraded RAID set.

Availability. To avoid a drop in user perceived performance and violation of SLAs, the user response time during reconstruction must be significantly reduced. WorkOut strives to achieve this goal by significantly reducing, if not eliminating, the contention between external user I/O requests and internal reconstruction requests, by outsourcing I/O workloads to a surrogate RAID set.

Extensibility. Since I/O intensity affects the performance of not only the reconstruction process but also other background support RAID tasks, such as re-synchronization and disk scrubbing, the idea of WorkOut should be readily extendable to improve the performance of these RAID tasks.

Flexibility. Due to the high cost and inconvenience involved in modifying the organization of an existing RAID, it is desirable to completely avoid such modification and instead utilize a separate surrogate RAID set judiciously and flexibly. In the WorkOut design, the surrogate RAID set can be a dedicated RAID1 set, a dedicated RAID5 set or a live RAID set that uses the free space of another operational (live) RAID set. Using a RAID as the surrogate set ensures that the redirected write data is safe-guarded with redundancy, thus guaranteeing the consistency of the redirected data. How to choose an appropriate surrogate RAID set is based on the requirements on overhead, performance, reliability, and maintainability and trade-offs between them.

3.2 WorkOut architecture overview

Figure 3 shows an overview of WorkOut’s architecture. In our design, WorkOut is an augmented module to the RAID controller software operating underneath the storage cache in a system with multiple RAID sets. WorkOut interacts with the reconstruction module, but is implemented independently of it. WorkOut can be incorporated into any RAID controller software, including various reconstruction approaches, and also other background support RAID tasks. In this paper, we focus on the reconstruction process and a discussion on how WorkOut works with some other background support RAID tasks can be found in Section 4.7.

WorkOut consists of five key functional components: Administration Interface, Popular Data Identifier, Surrogate Space Manager, Request Redirector and Reclaimer, as shown in Figure 3. *Administration Interface* provides an interface for system administrators to configure the WorkOut design options. *Popular Data Identifier*

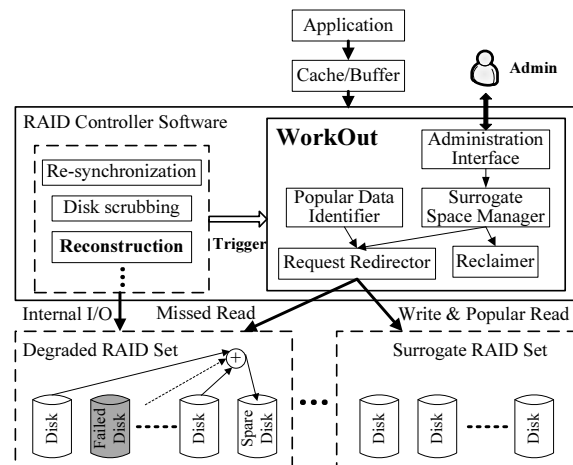


Figure 3: An architecture overview of WorkOut.

is responsible for identifying the popular read data. *Request Redirector* is responsible for redirecting all write requests and popular read requests to the surrogate RAID set, while *Reclaimer* is responsible for reclaiming all redirected write data back after the reconstruction process completes. *Surrogate Space Manager* is responsible for allocating and managing a space on the surrogate RAID set for each current reconstruction process and controlling the data layout of the redirected data in the allocated space.

WorkOut is automatically activated by the reconstruction module when the reconstruction thread is initiated and de-activated when the reclaim process completes. In other words, WorkOut is active throughout the entire reconstruction and reclaim periods. Moreover, the reclaim thread is triggered by the reconstruction module when the reconstruction process completes.

In WorkOut, the idea of a dedicated surrogate RAID set is targeted at a typical data center where the surrogate set can be shared by multiple degraded RAID sets on either a space-division or a time-division basis. The degraded RAID set and surrogate RAID set are not a one-to-one mapping. The device overhead is incurred only during reconstruction of a degraded RAID set and typically amounts to a small fraction of the surrogate set capacity. For example, extensive experiments on our prototype implementation of WorkOut show that, of all the traces and benchmarks, no more than 4% of the capacity of a 4-disk surrogate RAID5 set is used during the WorkOut reconstruction.

Based on the pre-configured parameters by system administrators, the Surrogate Space Manager allocates a disjoint space for each degraded RAID set that requests a surrogate RAID set, thus preventing the redirected data from being overwritten by redirected data from other degraded RAID sets. Noticeably, the space allocated to a degraded RAID set is not fixed and can be expanded. For

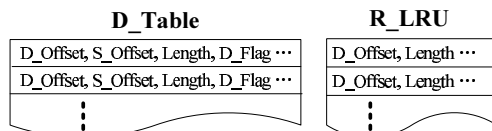


Figure 4: Data structures of WorkOut.

example, the Surrogate Space Manager first allocates an estimated space required for a typical degraded RAID set and, if the allocated space is used up to a preset threshold (e.g., 90%), it will allocate some extra space to this RAID set. In this paper, we mainly consider the scenario where there is at most one degraded RAID set at any given time. Implementing and evaluating WorkOut in a large-scale storage system with multiple concurrent degraded RAID sets are work in process.

3.3 The WorkOut organization and algorithm

WorkOut relies on two key data structures to redirect requests and identify popular data, namely, *D_Table* and *R_LRU*, as shown in Figure 4. *D_Table* contains the log of all redirected data, including the following four important variables. *D_Offset* and *S_Offset* indicate the offsets of the redirected data in the degraded RAID set and the surrogate RAID set, respectively. *Length* indicates the length of the redirected data and *D_Flag* indicates whether it is the redirected write data from the user application (*D_Flag* is set to be true) or the redirected read data from the degraded RAID set (*D_Flag* is set to be false). *R_LRU* is an LRU list that stores the information (*D_Offset* and *Length* of read data) of the most recent read requests. Based on *R_LRU*, popular read data can be identified and redirected to the surrogate RAID set.

WorkOut focuses on outsourcing user I/O requests during reconstruction and does not modify the reconstruction algorithm. How to perform the reconstruction process remains the responsibility of the reconstruction module and depends on the specific reconstruction algorithm, and is thus not described further in this paper.

During reconstruction, all write requests are redirected to the surrogate RAID set after determining whether they should overwrite their previous location or write to a new location according to *D_Table*. Whereas, for each read request, *D_Table* is first checked to determine whether the read data is in the surrogate RAID set. If the read request does not hit *D_Table*, it will be served by the degraded RAID set. If it hits *R_LRU*, the read data is considered popular and redirected to the surrogate RAID set, and the corresponding data information is inserted into *D_Table*. If the entire targeted read data is already in the surrogate RAID set, the read request will be served by the surrogate RAID set. Otherwise, if only a portion of the read data is in the surrogate RAID set, i.e., it partially hits *D_Table*, the read request will be split and served by both the sets. In order to achieve better performance, the

redirected data is laid out sequentially like LFS [33] in the allocated space on the surrogate RAID set.

The redirected write data is only temporarily stored in the surrogate RAID set and thus should be reclaimed back to the newly recovered RAID set (i.e., the formerly degraded RAID set) after the reconstruction process completes. To ensure data consistency, the log of reclaimed data is deleted from *D_Table* after the write succeeds. Since the redirected read data is already in the degraded RAID set, it need not be reclaimed as long as logs of such data are deleted from *D_Table* to indicate that the data in the surrogate RAID set is invalid. In order not to affect the performance of the newly recovered RAID set, the priority of the reclaim process is set to be the lowest, which will not affect the reliability of the redirected data as explained in Section 5.

During the reclaim period, all requests on the newly recovered RAID set must be checked carefully in *D_Table* to ensure data consistency. If a write request hits *D_Table* and its *D_Flag* is true, meaning that it will rewrite the old data that is still in the surrogate RAID set, the corresponding log in *D_Table* must be deleted after writing the data to the correct location on the newly recovered RAID set, to prevent the new write data from being overwritten by the reclaimed data. In addition, if a read request hits *D_Table* and its *D_Flag* is true, meaning that the up-to-date data of the read request has not been reclaimed back, the read request will be served by the surrogate RAID set.

3.4 Design choices

WorkOut can redirect data to different persistent configurations of storage devices, such as a dedicated surrogate RAID1 set, a dedicated surrogate RAID5 set and a live surrogate RAID set.

A dedicated surrogate RAID1 set. In this case, WorkOut stores the redirected data in *two mirroring disks*, namely, a dedicated surrogate RAID1 set. The advantage of this design option is its high reliability, simple space management and moderate device overhead (i.e., 2 disks), while its disadvantage is obvious: relatively low performance gain due to the lack of I/O parallelism.

A dedicated surrogate RAID5 set. In favor of reliability and performance (access parallelism), a dedicated surrogate RAID5 set with *several disks* can be deployed to store the redirected data. The space management is simple while the device overhead (e.g., 4 disks) is relatively high.

A live surrogate RAID set. WorkOut can utilize the free space of another live surrogate RAID set in a large-scale storage system consisting of multiple RAID sets and does not incur any additional device overhead that the first two design options cannot avoid. In this case, WorkOut gains high reliability owing to its redun-

Optional surrogate RAID set	Device Overhead	Performance	Reliability	Maintainability
A dedicated surrogate RAID1 set	medium	medium	high	simple
A dedicated surrogate RAID5 set	high	high	high	simple
A live surrogate RAID set	low	low	medium-high	complicated

Table 1: Characteristic comparisons of three optional surrogate RAID sets used in WorkOut.

dancy, but requires complicated maintenance. Due to the contention between the redirected requests from the degraded RAID set and the native I/O requests targeted at the live surrogate RAID set, the performance in this case is lower than that in the former two design options.

The three design options are all feasible and can be made available for system administrators to choose from through the Administration Interface based on their characteristics and tradeoffs, as summarized in Table 1. In this paper, the prototype implementation and performance evaluations are centered around the dedicated surrogate RAID5 set, although sample results from the other two design choices are also given to show the quantitative differences among them.

3.5 Data consistency

Data consistency in WorkOut includes two aspects: (1) Redirected data must be reliably stored in the surrogate RAID set, (2) The key data structures should be safely stored until the reclaim process completes.

First, in order to avoid data loss caused by a disk failure in the surrogate RAID set, all redirected write data in the surrogate RAID set should be protected by a redundancy scheme, such as RAID1 or RAID5. To simplify the design and implementation, the redirected read data is stored in the same manner as the redirected write data. If a disk failure in the surrogate RAID set occurs, data will no longer be redirected to the surrogate RAID set and the write data that was already redirected should be reclaimed back to the degraded RAID set or redirected to another surrogate RAID set if possible. Our prototype implementation adopts the first option. We will analyze the reliability of WorkOut in Section 5.

Second, since we must ensure never to lose the contents of D_Table during the entire period when WorkOut is activated, it is stored in a NVRAM to prevent data loss in the event of a power supply failure. Fortunately, D_Table is in general very small (see Section 4.8) and thus will not incur significant hardware cost. Moreover, since the performance of battery-backed RAM, a *de facto* standard form of NVRAM for storage controllers [8, 15, 16], is roughly the same as the main memory, the write penalty due to D_Table updates can be negligible.

4 Performance Evaluations

In this section, we evaluate the performance of prototype implementation of WorkOut through extensive trace-driven and benchmark-driven experiments.

4.1 Prototype implementation

We have implemented WorkOut by embedding it into the Linux Software RAID (MD) as a built-in module. In order not to impact the RAID performance in normal mode, WorkOut is activated only when the reconstruction process is initiated. During reconstruction, WorkOut tracks user I/O requests in the *make_request* function and issues them to the degraded RAID set or the surrogate RAID set based on the request type and D_Table.

By setting the reconstruction bandwidth range, MD assigns different disk bandwidth to serve user I/O requests and reconstruction requests and ensures that the reconstruction speed is confined within the set range (i.e., between the minimum and maximum reconstruction bandwidth). For example, if the reconstruction bandwidth range is set to be the default of 1MB/s-200MB/s, MD will favor user I/O requests while ensuring that the reconstruction speed is at least 1MB/s. Under heavy I/O workloads, MD will keep the reconstruction speed at approximately 1MB/s but allows it to be much higher than 1MB/s when I/O intensity is low. At one extreme when there is no user I/O, the reconstruction speed will be roughly equal to the disk transfer rate (e.g., 78MB/s in our prototype system). Equivalently, the *minimum reconstruction bandwidth* of *X*MB/s (e.g., 1MB/s, 10MB/s, 100MB/s) refers to a reconstruction range of *X*MB/s-200MB/s in MD. When the minimum reconstruction bandwidth is set to 100MB/s, which is not achievable for most disks, MD utilizes any disk bandwidth available for the reconstruction process.

To better examine the WorkOut performance on existing RAID reconstruction algorithms, we incorporate WorkOut into MD's default reconstruction algorithm PR, and PRO-powered PR (PRO for short) that is also implemented in MD. PR (Pipeline Reconstruction) [22] takes advantage of the sequential property of track retrievals to pipeline the reading and writing processes. PRO (Popularity-based multi-threaded Reconstruction Optimization) [41, 42] allows the reconstruction process to rebuild the frequently accessed areas prior to other areas.

4.2 Experimental setup and methodology

We conduct our performance evaluation of WorkOut on a platform of server-class hardware with an Intel Xeon 3.0GHz processor and 1GB DDR memory. We use 2 Highpoint RocketRAID 2220 SATA cards to house 15 Seagate ST3250310AS SATA disks. The rotational speed of these disks is 7200 RPM, with a sustained trans-

Trace	Trace Characteristic		
	Write Ratio	IOPS	Aver. Req. Size(KB)
Fin1	67.18%	69	6.2
Fin2	17.61%	125	2.2
Web	0%	113	15.1

Table 2: The trace characteristics.

fer rate of 78MB/s, and the individual disk capacity is 250GB. A separate IDE disk is used to house the operating system (Fedora Core 4 Linux, kernel version 2.6.11) and other software (MD and mdadm). For the footprint of the workloads, we limit the capacity of each disk to 10GB in the experiments. In our prototype implementation, the main memory is used to substitute a battery-backed RAM for simplicity.

Generally speaking, there are two models for trace replay: open-loop and closed-loop [26, 35]. The former has the potential to overestimate the user response time measure since the I/O arrival rate is independent of the underlying system and thus can cause the request queue (and hence the queuing delays) to grow rapidly when system load is high. The opposite is true for closed systems as the I/O arrival rate is dictated by the processing speed of the underlying system and the request queue is generally limited in length (i.e., equal to the number of independent request threads). In this paper, we use both an open-loop model (trace replay with RAIDmeter [41]) and a closed-loop model (TPC-C-like benchmark [43]) to evaluate the performance of WorkOut.

The traces used in our experiments are obtained from the Storage Performance Council [29, 39]. The two financial traces (Fin1 and Fin2) were collected from OLTP applications running at a large financial institution and the WebSearch2 trace (or Web) was collected from a machine running a web search engine. The three traces represent different access patterns in terms of write ratio, IOPS and average request size, as shown in Table 2. The write ratio of the Fin1 trace is the highest, followed by the Fin2 trace. The read-dominated Web trace exhibits the strongest locality in its access pattern. Since the request rate in the Web trace is too high to be sustained by our degraded RAID set, we only use one part of it that is attributed to device zero while the part due to devices one and two is ignored.

Since the three traces have very limited footprints, that is the user I/O requests are congregated on a small part of the RAID set (e.g., less than 10% of an 8-disk RAID5 set for the Fin1 trace), their replays may not realistically represent a typical reconstruction scenario where user requests may be spread out over the entire disk address space. To fully and evenly cover the address space of the RAID set, we scale up the address coverage of the I/O requests by multiplying the address of each request with an appropriate scaling factor (constant) without changing the size of each request. While the main adverse

impact of this trace scaling is likely to be on those requests that are originally sequential but can subsequently become non-sequential after scaling, the percentage of such sequential requests in the three traces of this study is relatively small at less than 4% [45]. Thus, we believe that the adverse impact of the trace scaling is rather limited in this study and far outweighed by the benefits of the scaling that attempts to represent a more realistic reconstruction scenario. We find in our experiments that the observed trends are similar for the original and the scale trace, suggesting that neither is likely to generate noticeably different conclusions for the study. Nevertheless, we choose to present the results of the latter for the aforementioned reasons.

The trace replay tool is RAIDmeter [41, 42] that replays traces at block-level and evaluates the user response time of the storage device. The RAID reconstruction performance is evaluated in terms of the following two metrics: reconstruction time and average user response time during reconstruction.

The TPC-C-like benchmark is implemented with TPCC-UVA [31] and the Postgres database. It generates mixed transactions based on the TPC-C specification [43]. 20 warehouses are built on the Postgres database with the ext3 file system on the degraded RAID set. Transactions, such as PAYMENT, NEW_ORDER and DELIVERY, generate read and write requests. To evaluate the WorkOut performance, we compare the transaction rates (*transactions per minute*) that are generated at the end of the benchmark execution.

4.3 Trace-driven evaluations

We first conduct experiments on an 8-disk RAID5 set with a stripe unit size of 64KB while running PR, PRO and WorkOut-powered PR and PRO respectively. Table 3 and Table 4, respectively, show the reconstruction time and average user response time under the minimum reconstruction bandwidth of 1MB/s, driven by the three traces. We configure a 4-disk dedicated RAID5 set with a stripe unit size of 64KB as the surrogate RAID set to boost the reconstruction performance of the 8-disk degraded RAID set.

From Table 3, one can see that WorkOut speeds up the reconstruction time by a factor of up to 5.52, 1.64 and 1.30 for the Fin1, Fin2 and Web traces, respectively. The significant improvement achieved on the Fin1 trace, with a reconstruction time of 203.1s vs. 1121.8s for PR and 188.3s vs. 1109.6s for PRO, is due to the fact that 84% of requests (69% of writes plus 15% of reads) are redirected away from the degraded RAID set (see Figure 5), which enables the speed of the on-line reconstruction to approach that of the off-line counterpart. In our experiments, the off-line reconstruction time is 136.4 seconds for PR on the same platform. Moreover, WorkOut out-

Traces	Reconstruction Time (second)						
	Off-line	PR	WorkOut+PR	speedup	PRO	WorkOut+PRO	speedup
Fin1	136.4	1121.8	203.1	5.52	1109.6	188.3	5.89
Fin2		745.2	453.3	1.64	705.8	431.2	1.64
Web		9935.6	7623.2	1.30	9888.3	7851.4	1.26

Table 3: The reconstruction time results.

Traces	Average User Response Time (millisecond)							
	Normal	Degraded	PR	WorkOut+PR	speedup	PRO	WorkOut+PRO	speedup
Fin1	7.9	9.5	12.7	4.4	2.87	9.8	4.6	2.15
Fin2	8.1	13.4	25.8	9.7	2.66	23.0	10.2	2.25
Web	18.5	27.0	38.6	28.3	1.36	35.6	29.1	1.22

Table 4: The average user response time results.

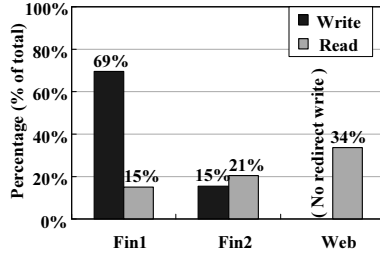


Figure 5: Percentage of redirected requests for WorkOut, under the minimum reconstruction bandwidth of 1MB/s.

sources 36% and 34% of user I/O requests away from the degraded RAID set for the Fin2 and Web traces, which is much fewer than that for the Fin1 trace, thus reducing the reconstruction time accordingly.

Table 4 shows that, compared with PR, WorkOut speeds up the average user response time by a factor of up to 2.87, 2.66 and 1.36 for the Fin1, Fin2 and Web traces, respectively. For Fin1 and Fin2, the average user response times during reconstruction under WorkOut are even better than that in the normal or degraded period. The reasons why WorkOut achieves significant improvement on user response times are threefold. First, a significant amount of requests are redirected away from the degraded RAID set, as shown in Figure 5. The response times of redirected requests are no longer affected by the reconstruction process that competes for the available bandwidth with user I/O requests on the degraded RAID set. Second, redirected data is laid out sequentially in the surrogate RAID set, thus further speeding up the user response time. Third, since many requests are outsourced, the I/O queue on the degraded RAID set is shortened accordingly, thus reducing the response times of the remaining I/O requests served by the degraded RAID set. Therefore, the average user response time with WorkOut is significantly lower than that without WorkOut, especially for the Fin1 trace.

Table 3 and Table 4 show that WorkOut-powered PRO performs similarly to WorkOut-powered PR. The reason is that WorkOut redirects all write requests and popular read requests to the surrogate RAID set, thus reducing

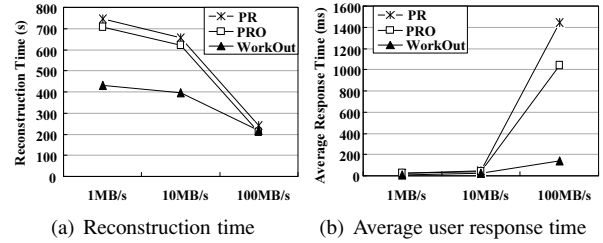


Figure 6: Comparisons of reconstruction times and average user response times with respect to different minimum reconstruction bandwidth (1MB/s, 10MB/s and 100MB/s) driven by the Fin2 trace.

the degree of popularity of I/O workloads retained on the degraded RAID set that can be exploited by PRO. Based on this observation, in the following experiments, we only compare WorkOut-powered PR (*short for WorkOut*) with PR and PRO.

4.4 Sensitivity study

WorkOut's performance is likely influenced by several important factors, including the available reconstruction bandwidth, the size of the degraded RAID set, the stripe unit size, and the RAID level. Due to lack of space, we limit our study of these parameters to the Fin2 trace. Other traces show similar trends as the Fin2 trace.

Reconstruction bandwidth. To evaluate how the minimum reconstruction bandwidth affects reconstruction performance, we conduct experiments that measure reconstruction times and average user response times as a function of different minimum reconstruction bandwidth, 1MB/s, 10MB/s and 100MB/s, respectively. Figure 6 plots the experimental results on an 8-disk RAID5 set with a stripe unit size of 64KB.

Figure 6(a) shows that WorkOut speeds up the reconstruction time more significantly with a lower minimum reconstruction bandwidth than with a higher one. The reason is that the reconstruction process already exploits all available disk bandwidth when the reconstruction bandwidth is higher, thus leaving very small room for the reconstruction time to be further improved.

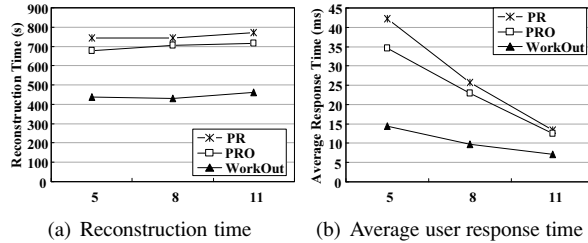


Figure 7: Comparisons of reconstruction times and average user response times with respect to the number of disks (5, 8, 11) driven by the Fin2 trace.

From Figure 6(b), in contrast, the user response time increases rapidly with the increasing minimum reconstruction bandwidth for both PR and PRO, but much more slowly for WorkOut. WorkOut speeds up the user response time significantly, by a factor of up to 10.2 and 7.38 over PR and PRO, respectively, when the minimum reconstruction bandwidth is set to 100MB/s. From this viewpoint, the user response time with WorkOut is much less sensitive to the minimum reconstruction bandwidth than that without WorkOut. In other words, if the reconstruction bandwidth is set very high or the storage system is reliability-oriented, that is the reconstruction process is given more bandwidth to favor the system reliability, the user response time improvement by WorkOut will be much more significant. Moreover, the user response time during reconstruction for PR and PRO is so long that it will likely violate SLA and thus become unacceptable to end users.

Number of disks. To examine the sensitivity of WorkOut to the number of disks of the degraded RAID set, we conduct experiments on RAID5 sets consisting of different numbers of disks (5, 8 and 11) with a stripe unit size of 64KB under the minimum reconstruction bandwidth of 1MB/s. Figure 7 shows the experimental results for PR, PRO and WorkOut.

Figure 7(a) and Figure 7(b) show that for all three approaches, the reconstruction time increases and the user response time decreases for higher number of disks in the degraded RAID set. The reason is that more disks in a RAID set imply not only a larger RAID group size and thus more disk read operations to reconstruct a failed drive, but also higher parallelism for the I/O process. However, WorkOut is less sensitive to the number of disks than PR and PRO.

Stripe unit size. To examine the impact of the stripe unit size, we conduct experiments on an 8-disk RAID5 set with stripe unit sizes of 16KB and 64KB, respectively. The experimental results show that WorkOut outperforms PR and PRO in reconstruction time as well as average user response time for both stripe unit sizes. Moreover, the reconstruction times and average user re-

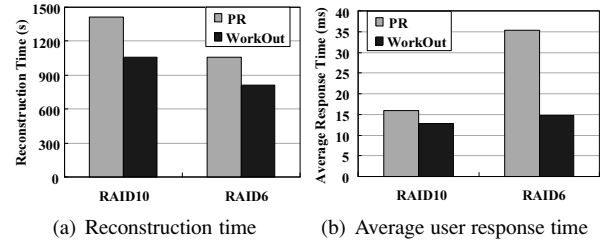


Figure 8: Comparisons of reconstruction times and average user response times with respect to different RAID levels (10, 6) driven by the Fin2 trace.

sponse times of WorkOut are almost unchanged, suggesting that WorkOut is not sensitive to the stripe unit size.

RAID level. To evaluate WorkOut with different RAID levels, we conduct experiments on a 4-disk RAID10 set and an 8-disk RAID6 set with the same stripe unit size of 64KB under the minimum reconstruction bandwidth of 1MB/s. In the RAID6 experiments, we measure the reconstruction performance when two disks fail concurrently.

From Figure 8, one can see that WorkOut speeds up both the reconstruction times and average user response times for the two sets. The difference in the amount of performance improvement seen for RAID10 and RAID6 is caused by the different user I/O intensities, since the RAID10 and RAID6 sets have different numbers of disks. The user I/O intensity on individual disks in the RAID10 set is higher than that in the RAID6 set, thus leading to longer reconstruction times.

On the other hand, since each read request to the failed disks in a RAID6 set must wait for its data to be rebuilt on-the-fly, the user response time is severely affected for PR, while this performance degradation is significantly lower under WorkOut due to its external I/O outsourcing. For the RAID10 set, however, the situation is quite different. Since the read data can be directly returned from the surviving disks, WorkOut provides smaller improvements in user response time for RAID10 than for RAID6.

4.5 Different design choices for the surrogate RAID set

All experiments reported up to this point in this paper adopt a dedicated surrogate RAID5 set. To examine the impact of different types of surrogate RAID set on the WorkOut performance, we also conduct experiments with a dedicated surrogate RAID1 set (two mirroring disks) and a live surrogate RAID set (replaying the Fin1 trace on a 4-disk RAID5 set). Similar to the experiments conducted in the PARAID [46] and write off-loading [27] studies, we reserve the 10% portion of storage space at the end of the live RAID5 set to store data redirected by WorkOut. The degraded RAID set is an 8-disk RAID5

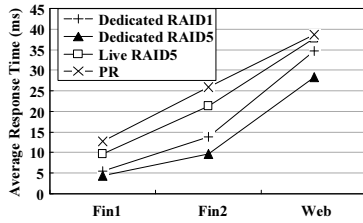


Figure 9: A comparison of average user response times for different types of surrogate RAID set.

set with a stripe unit size of 64KB and under the minimum reconstruction bandwidth of 1MB/s.

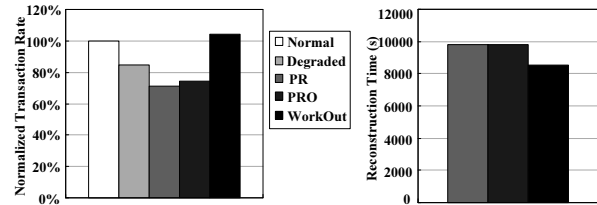
The experimental results show that the reconstruction times achieved by WorkOut are almost the same for the three types of surrogate RAID set and outperform PR as expected, shown in Table 3, since WorkOut outsources the same amount of requests during reconstruction. The results for user response times are somewhat different, as shown in Figure 9. The dedicated surrogate RAID5 set results in the best user response times for the three traces.

From Figure 9, one can see that the dedicated surrogate RAID sets (both RAID1 and RAID5) outperform the live surrogate RAID set in user response time. The reason is the contention between the native I/O requests and the redirected requests in the live surrogate RAID set. Serving the native I/O requests not only increases overload on the surrogate RAID set, compared with the dedicated surrogate RAID set, but also destroys some of the sequentiality in LFS style writes. The redirected requests also increase the overall I/O intensity on the live surrogate RAID set and affect its performance. Our experimental results show that the performance impact on the live surrogate RAID set is 43.9%, 23.6% and 36.8% on average when the degraded RAID set replays the Fin1, Fin2 and Web traces, respectively. The experimental results are consistent with the comparisons in Table 1.

4.6 Benchmark-driven evaluations

In addition to trace-driven experiments, we also conduct experiments on an 8-disk RAID5 set with a stripe unit size of 64KB under the minimum reconstruction bandwidth of 1MB/s, driven by a TPC-C-like benchmark.

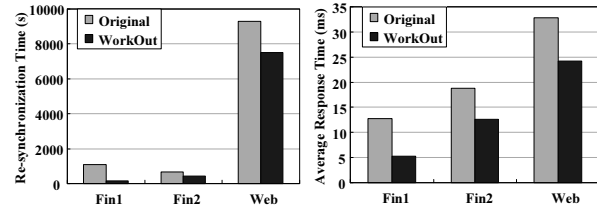
From Figure 10(a), one can see that PRO performs almost the same as PR due to the random access characteristics of the TPC-C-like benchmark. Since WorkOut outsources all write requests that are generated by the transactions, both the degraded RAID set and surrogate RAID set serve the benchmark application, thus increasing the transaction rate. WorkOut outperforms PR and PRO in terms of transaction rate, with an improvement of 46.6% and 36.9% respectively. It also outperforms the original system in the normal mode (the normalized baseline) and the degraded mode, with an improvement of 4.0% and 22.6% respectively.



(a) Transaction Rate

(b) Reconstruction time

Figure 10: Comparisons of reconstruction times and transaction rates driven by the TPC-C-like benchmark.



(a) Re-synchronization time

(b) Average user response time

Figure 11: Comparisons of re-synchronization times and average user response times during re-synchronization.

On the other hand, since the TPC-C-like benchmark is highly I/O intensive, all disks in the RAID set are driven to saturation, thus the reconstruction speed is kept at around its minimum allowable bandwidth of 1MB/s for PR and PRO. As shown in Figure 10(b), the reconstruction times for PR and PRO are similar, at 9835 seconds and 9815 seconds, respectively, while that for WorkOut is 8526 seconds, with approximately 15% improvement over PR and PRO. WorkOut gains much less in reconstruction time with the benchmark-driven experiments than with the trace-driven experiments. The main reason lies in the fact that the very high I/O intensity of the benchmark application constantly pushes the RAID set to operate at or close to its saturation point, leaving very little disk bandwidth for the reconstruction process even with some of the transaction requests being outsourced to the surrogate RAID set.

4.7 Re-synchronization with WorkOut

To demonstrate how WorkOut optimizes other background support RAID tasks, such as RAID re-synchronization, we conduct experiments on an 8-disk RAID5 set with a stripe unit size of 64KB under the minimum re-synchronization bandwidth of 1MB/s, driven by the three traces. We configure a dedicated 4-disk RAID5 set with a stripe unit size of 64KB as the surrogate RAID set. The experimental results of re-synchronization times and average user response times during re-synchronization are shown in Figure 11(a) and Figure 11(b), respectively.

Although the re-synchronization process performs somewhat differently from the reconstruction process,

re-synchronization requests also compete for the disk resources with user I/O requests. By redirecting a significant amount of user I/O requests away from the RAID set during re-synchronization, WorkOut can reduce both the re-synchronization times and user response times. The results are very similar to that in the reconstruction experiments, so are the reasons behind them.

4.8 Overhead analysis

Device overhead. WorkOut is designed for use in a large-scale storage system consisting of many RAID sets that share one surrogate RAID composed of spare disks. In such an environment, the device overhead introduced by WorkOut is small given that a single surrogate RAID can be shared by many production RAID sets. Nevertheless, for a small-scale storage system composed of only one or two RAID sets with few hot spare disks, the device overhead of a dedicated surrogate RAID set in WorkOut cannot be ignored. In this case, to be cost-effective, we recommend the use of a dedicated surrogate RAID1 set instead of a dedicated surrogate RAID5 set, since the device overhead of the former (i.e., 2 disks) is lower than that of the latter (e.g., 4 disks in our experiments).

To quantify the cost-effectiveness of WorkOut in this resource-restricted environment, we conduct experiments and compare the performance of WorkOut (8-disk data RAID5 set plus 4-disk surrogate RAID5 set) with that of PR (12-disk data RAID5 set), i.e., we use the same number of disks in both systems. Experiments are run under the minimum reconstruction bandwidth of 1MB/s and driven by the Fin2 trace. The results show that WorkOut speeds up the reconstruction time of PR significantly, by a factor of 1.66. The average user response time during reconstruction achieved by WorkOut is 16.5% shorter than that achieved by PR, while the average user response time during the normal period in the 8-disk RAID5 set is 20.1% longer than that in the 12-disk RAID5 set due to the reduced access parallelism of the former. In summary, we can conclude that WorkOut is cost-effective in both large-scale and small-scale storage systems.

Memory overhead. To prevent data loss, WorkOut uses non-volatile memory to store D_Table, thus incurring extra memory overhead. The amount of memory consumed is largest when the minimum reconstruction bandwidth is set to 1MB/s, since in this case the reconstruction time is the longest and the amount of redirected data is the largest. In the above experiments on the RAID5 set with individual disk capacity of 10GB, the maximum memory overheads are 0.14MB, 0.62MB and 1.69MB for the Fin1, Fin2 and Web traces, respectively. However, the memory overhead incurred by WorkOut is only temporary and will be removed after the reclaim process completes. With the rapid increase in memory

size and decrease in cost of non-volatile memories, this memory overhead is arguably reasonable and acceptable to end users.

Implementation overhead. WorkOut contains 780 lines of added or modified code to the source code of the Linux software RAID (MD), with most lines of code added to md.c and raidx.c while 37 lines of data structure code added to md.k.h and raidx.h. Since most of the added code is independent of the underlying RAID layout, they are easy to be shared by different RAID levels. Moreover, the added code is independent of the reconstruction module, so it is easy to adapt the code for use with other background support RAID tasks. All that needs to be done is modifying the corresponding flag that triggers WorkOut. Due to the independent implementation of the WorkOut module, it is portable to other software RAID implementations in other operating systems.

5 Reliability Analysis

In this section, we adopt the MTTFDL metric to estimate the reliability of WorkOut. We assume that disk failures are independent events following an exponential distribution of rate μ , and repairs follow an exponential distribution of rate ν . For simplicity, we do not consider the latent sector error in the system model.

According to the conclusion about the reliability of RAID5 [50], MTTFDL of an 8-disk RAID5 set achieved by PR and PRO is:

$$MTTFDL_{RAID5-8} = \frac{15\mu + \nu}{56\mu^2} \quad (1)$$

Figure 12 shows the state transition diagram for a WorkOut-enabled storage system configuration consisting of an 8-disk data RAID5 set and a 4-disk surrogate RAID5 set. Note that by design WorkOut always reclaims the redirected write data from the surrogate RAID set upon a surrogate disk failure. Once the reclaim process is completed there is no more valid data of the degraded RAID set on the surrogate RAID set. Therefore, there is no need to reconstruct data on the failed disk of the surrogate RAID set. This means that the degraded surrogate RAID set can be recovered by simply replacing the failed disk with a new one, resulting in a newly recovered operational 4-disk surrogate RAID5 set ready to be used by the 8-disk degraded data RAID5 set. As a result, the state transition diagram only shows the reclaim process but not the reconstruction process of the 4-disk surrogate RAID5 set.

State <0> represents the normal state of the system when its 8 data disks are all operational. A failure of any of the 8 data disks would bring the system to state <1> and a subsequent failure of any of the remaining 7 data disks would result in data loss. A failure of any of the 4 surrogate disks in state <0> does not affect the system

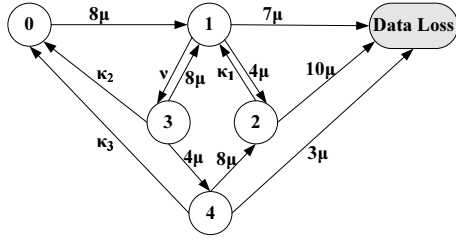


Figure 12: State transition diagram for a WorkOut-enabled storage system configuration consisting of an 8-disk RAID5 set and a 4-disk surrogate RAID5 set. *Note: The 4-disk surrogate RAID5 set does not need to reconstruct the data on the failed disk as long as the redirected write data in the surrogate RAID5 set is reclaimed back to the 8-disk degraded data RAID set.*

reliability of the 8-disk RAID5 set as long as the redirected write data on the former is reclaimed back to the latter and thus it is omitted from the state transition diagram. In state $\langle 1 \rangle$, a failure of any of the 4 surrogate disks would bring the system to state $\langle 2 \rangle$. A second failure in either the 8-disk data RAID5 set (1 out of 7) or the 4-disk surrogate RAID5 set (1 out of 3) in state $\langle 2 \rangle$ would result in data loss. In state $\langle 2 \rangle$, WorkOut reclaims the redirected write data back to the 8-disk data RAID set, which brings the system back to state $\langle 1 \rangle$ and follows an exponential distribution of rate κ_1 . This transition implicitly assumes that, while redirected write data is being reclaimed from the surrogate set to the data set, the reconstruction process on the latter is temporarily suspended. This simplifying assumption is justifiable and will not affect the result noticeably since the reclaim time is much shorter than the reconstruction time on the 8-disk data RAID set. Finishing the reconstruction process of the 8-disk data RAID5 set would bring the system from state $\langle 1 \rangle$ to state $\langle 3 \rangle$, where the redirected write data has not been reclaimed. Then finishing the reclaim process would bring it back to state $\langle 0 \rangle$, which follows an exponential distribution of rate κ_2 . In state $\langle 3 \rangle$, a failure of any of the 8 data disks would bring the system to state $\langle 1 \rangle$, and a failure of any of the 4 surrogate disks would bring the system to state $\langle 4 \rangle$, where the redirected write data is not protected by redundancy. In state $\langle 4 \rangle$, WorkOut also reclaims the redirected write data back to the 8-disk data RAID set, which bring the system back to state $\langle 0 \rangle$ and follows an exponential distribution of rate κ_3 . In state $\langle 4 \rangle$, a failure of any of the 8 data disks would bring the system to state $\langle 2 \rangle$ and a second disk failure in the 4-disk surrogate RAID5 set would result in data loss.

Since κ_1 , κ_2 and κ_3 all represent the rate at which the redirected write data is reclaimed, it is reasonable to assume that they are equal to a fixed reclaim rate κ , since the amount of redirected write data should be roughly the

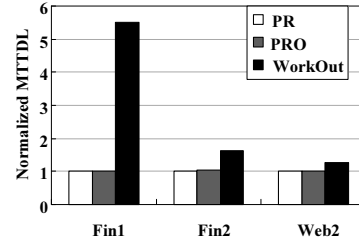


Figure 13: Comparisons of the mean times to data loss. *Note: The normalized baselines are the MTDLs achieved by PR driven by the three traces respectively.*

same and the rate of transferring this data should also be the same under reasonable circumstances for all the three cases. Since the expression for MTDL of Figure 12 is too complex (ratio of two large polynomials) to be displayed here, we present the computed values of MTDL in the following figure instead.

Figure 13 plots comparisons of MTDLs achieved by PR, PRO and WorkOut, which are normalized to the MTDLs achieved by PR driven by the three traces respectively. The disk failure rate μ is assumed to be one failure every one hundred thousand hours, which is a conservative estimate to the values quoted by disk manufactures. Disk repair times, when the individual disk capacity is 250GB, are 25 times the results listed in Table 3, where the capacity of each disk is limited to 10GB. κ is assumed to be equal to the corresponding ν , which is actually overestimated. From Figure 13, one can see that WorkOut increases MTDL and improves reliability with the decreasing MTTR, especially for the write-intensive trace (i.e., Fin1). Moreover, if we alter κ to be several times smaller or larger than ν , the black bar remains almost unchanged, suggesting that the reclaim time does not affect the reliability of the RAID system and thus can be excluded from the reconstruction time.

6 Related Work

Reconstruction algorithms and task scheduling. A large number of different approaches for improving reconstruction performance have been studied. Some of these approaches focus on improved RAID reconstruction algorithms, such as DOR (Disk-Oriented Reconstruction) [12], PR [22], PRO [41] and others [3, 13, 14, 20, 38, 47, 48, 49]. Other approaches focus on better data layout in a RAID set [13, 47, 49]. Moreover, many task scheduling techniques have been proposed to optimize the background applications [25, 40, 44].

While all the the above algorithms focus on improving performance by optimizing the organization of work *within a single RAID set*, our work takes a different approach. The goal behind WorkOut is to increase performance during reconstruction by outsourcing I/O workloads away from the degraded RAID set. Importantly,

WorkOut is orthogonal to and can further improve the above techniques.

Data migration. Our study is related in spirit to write off-loading [27, 28] and data migration [2, 19, 21, 24, 46] techniques, but with distinctively different characteristics. Write off-loading [27] redirects writes from one volume to another, to prolong the idle period for one volume allowing the system to spin down disks for saving energy. Similar in spirit, Everest [28] off-loads writes from overloaded volumes to lightly loaded ones to improve performance during peaks.

Data migration [24] moves data from one storage device to another, e.g., for the purpose of load balancing (or load concentration), failure recovery, or system expansion. Data migration has been used in the context of improving energy efficiency PARAD [46], improving performance by data reallocation (e.g., in the products of EMC's Symmetrix family [2]), for read request offloading in Cuckoo [21] and for user-centric data migration in networked storage systems [19].

In contrast to write off-loading and data migration, WorkOut improves reconstruction performance by, *temporarily* redirecting writes and popular reads during reconstruction and reclaiming the redirected write data back to the newly recovery RAID set after the reconstruction process completes.

7 Future Work

WorkOut is an ongoing research project and we are currently exploring several directions for future work.

Extendibility. In addition to RAID reconstruction and re-synchronization, other background support RAID tasks, such as disk scrubbing and block-level backup and snapshot, could benefit from WorkOut. We plan to conduct detailed experiments to measure the impact of WorkOut on these tasks.

Flexibility. In the current implementation, we configure a reserved space instead of the free space on a live RAID set as the surrogate set, which can be impractical and inflexible. Utilizing the free space on a live RAID set at the file system level is complicated as the file system must be engaged to discover, assign, protect and manage the free space [37]. To make WorkOut more transparent to the file system, and more effectively utilize the free space on a live surrogate RAID set, it would be desirable for WorkOut to obtain the liveness information at the block level. We will explore the live block techniques and apply them in WorkOut to improve its performance.

8 Conclusion

In this paper, for significantly boosting RAID reconstruction performance, we propose *WorkOut* (I/O Workload Outsourcing) that outsources a significant amount of user I/O requests away from the degraded RAID set to a sur-

rogate RAID set during reconstruction. We present a lightweight prototype of WorkOut implemented in the Linux software RAID. In a detailed experimental evaluation, we demonstrate that, compared with the existing reconstruction algorithms PR and PRO, WorkOut significantly speeds up the reconstruction time and average user response time simultaneously. Moreover, we provide insights and guidance for storage system designers and administrators by exploiting three WorkOut design options based on their device overhead, performance, reliability, maintainability and trade-offs. Importantly, we demonstrate how WorkOut can be easily deployed to improve the performance of other background support RAID tasks such as re-synchronization.

Acknowledgments

We thank our shepherd Bianca Schroeder and the anonymous reviewers for their helpful comments. We also thank Lingfang Zeng, Jianxi Chen and Zhikun Wang for their feedback. This work is supported by the National Basic Research 973 Program of China under Grant No. 2004CB318201, the US NSF under Grant No. CCF-0621526, the China NSFC under Grant No. 60703046 and No. 60873028, the Program for New Century Excellent Talents in University No. NCET-04-0693 and No. NCET-06-0650, the Program for Changjiang Scholars and Innovative Research Team in University No. IRT-0725, the Programme of Introducing Talents of Discipline to Universities (111 Project) No. B07038, SRFDP of Education of China No. 20070487083, and HUST-SRF No.2007Q021B. The work of Lei Tian was done while he was working at the CSE Dept. of UNL.

References

- [1] M. Arlitt and C. Williamson. Web Server Workload Characterization: The Search for Invariants. In *SIGMETRICS'96*, May, 1996.
- [2] R. Arnan, E. Bachmat, T. K. Lam, and R. Michel. Dynamic Data Reallocation in Disk Arrays. *ACM Transactions on Storage*, 3(1), 2007.
- [3] E. Bachmat and J. Schindler. Analysis of Methods for Scheduling Low Priority Disk Drive Tasks. In *SIGMETRICS'02*, Jun. 2002.
- [4] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS'07*, Jun. 2007.
- [5] L. Cherkasova and G. Ciardo. Characterizing Temporal Locality and its Impact on Web Server Performance. Technical Report HPL-2000-82, Hewlett Packard Laboratories, Jul. 2000.
- [6] L. Cherkasova and M. Gupta. Analysis of Enterprise Media Server Workloads: Access Patterns, Locality, Content Evolution, and Rates of Change. *IEEE/ACM Transactions on Networking*, 12(5):781–794, Oct. 2004.
- [7] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *FAST'05*, Dec. 2005.
- [8] EMC storage products. <http://www.emc.com/products/category/storage.htm>.

- [9] G. Gibson. Reflections on Failure in Post-Terascale Parallel Computing. Keynote. In *ICPP'07*, Sep. 2007.
- [10] J. Gray. Rules of Thumb in Data Engineering. Keynote Address. In *ICDE'00*, Feb. 2000.
- [11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Fourth edition, 2006.
- [12] M. Holland. *On-Line Data Reconstruction in Redundant Disk Arrays*. PhD thesis, Carnegie Mellon University, Apr. 1994.
- [13] M. Holland and G. Gibson. Parity Declustering for Continuous Operation in Redundant Disk Arrays. In *ASPLOS'92*, Oct. 1992.
- [14] R. Hou, J. Menon, and Y. Patt. Balancing I/O Response Time and Disk Rebuild Time in a RAID5 Disk Array. In *HICSS'93*, 1993.
- [15] HP Disk Storage Systems. http://h18006.www1.hp.com/storage/disk_storage/index.html.
- [16] IBM Disk Storage Systems. <http://www-03.ibm.com/systems/storage/disk/>.
- [17] Iometer. <http://sourceforge.net/projects/iometer>.
- [18] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky. Are Disks the Dominant Contributor for Storage Failures? A Comprehensive Study of Storage Subsystem Failure Characteristics. In *FAST'08*, Feb. 2008.
- [19] S. Kang and A. L. N. Reddy. User-Centric Data Migration in Networked Storage Systems. In *IPDPS'08*, Apr. 2008.
- [20] H. H. Kari, H. K. Saikkonen, N. Park, and F. Lombardi. Analysis of repair algorithms for mirrored-disk systems. *IEEE Transactions on Reliability*, 46(2):193–200, 1997.
- [21] A. J. Klosterman and G. Ganger. Cukoo: Layered clustering for NFS. Technical Report CMU-CS-02-183, Carnegie Mellon University, Oct. 2002.
- [22] J. Y.B. Lee and J. C.S. Lui. Automatic Recovery from Disk Failure in Continuous-Media Servers. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):499–515, May. 2002.
- [23] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-Miner: Mining Block Correlations in Storage Systems. In *FAST'04*, Mar. 2004.
- [24] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: Online Data Migration with Performance Guarantees. In *FAST'02*, Jan. 2002.
- [25] C. R. Lumb, J. Schindler, G. R. Ganger, D. F. Nagle, and E. Riedel. Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives. In *OSDI'00*, Oct. 2000.
- [26] M. P. Mesnier, M. Wachs, R. R. Sambasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. O'Hallaron. //TRACE: Parallel Trace Replay with Approximate Causal Events. In *FAST'07*, Feb. 2007.
- [27] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *FAST'08*, Feb. 2008.
- [28] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling Down Peak Loads Through I/O Off-loading. In *OSDI'08*, Dec. 2008.
- [29] OLTP Application I/O and Search Engine I/O. UMass Trace Repository. <http://traces.cs.umass.edu/index.php/Storage/Storage>.
- [30] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD'88*, Jun. 1988.
- [31] J. Piernas, T. Cortes, and J. M. García. Tpc-c-uva: A free, open-source implementation of the tpc-c benchmark. <http://www.infor.uva.es/~diego/tpcc-uva.html>. 2005.
- [32] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *FAST'07*, Feb. 2007.
- [33] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *SOSP'92*, Feb. 1992.
- [34] B. Schroeder and G. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *FAST'07*, Feb. 2007.
- [35] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *NSDI'06*, May. 2006.
- [36] T. J. E. Schwarz, Q. Xin, E. L. Miller, D. D. E. Long, A. Hospodor, and S. Ng. Disk Scrubbing in Large Archival Storage Systems. In *MASCOTS'04*, Oct. 2004.
- [37] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block-Level. In *OSDI'04*, Dec. 2004.
- [38] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *FAST'04*, Mar. 2004.
- [39] Storage Performance Council. <http://www.storageperformance.org/home>.
- [40] E. Thereska, J. Schindler, J. Bucy, B. Salmon, C. R. Lumb, and G. R. Ganger. A framework for building unobtrusive disk maintenance applications. In *FAST'04*, Apr. 2004.
- [41] L. Tian, D. Feng, H. Jiang, K. Zhou, L. Zeng, J. Chen, Z. Wang, and Z. Song. PRO: A Popularity-based Multi-threaded Reconstruction Optimization for RAID-Structured Storage Systems. In *FAST'07*, Feb. 2007.
- [42] L. Tian, H. Jiang, D. Feng, Q. Xin, and X. Shu. Implementation and Evaluation of a Popularity-Based Reconstruction Optimization Algorithm in Availability-Oriented Disk Arrays. In *MSST'07*, Sep. 2007.
- [43] TPC-C specification. <http://www.tpc.org/tpcc/>.
- [44] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *FAST'07*, Feb. 2007.
- [45] M. Wang. *Performance Modeling of Storage Devices using Machine Learning*. PhD thesis, Carnegie Mellon University, Jan. 2006.
- [46] C. Weddle, M. Oldham, J. Qian, A. A. Wang, P. Reiher, and G. Kuenning. PARAID: The Gear-Shifting Power-Aware RAID. In *FAST'07*, Feb. 2007.
- [47] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *FAST'08*, Feb. 2008.
- [48] T. Xie and H. Wang. MICRO: A Multilevel Caching-Based Reconstruction Optimization for Mobile Storage Systems. *IEEE Transactions on Computers*, 57(10):1386–1398, 2008.
- [49] Q. Xin, E. L. Miller, and T. J. E. Schwarz. Evaluation of Distributed Recovery in Large-Scale Storage Systems. In *HPDC'04*, Jun. 2004.
- [50] Q. Xin, E. L. Miller, T. J. E. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin. Reliability Mechanisms for Very Large Storage Systems. In *MSST'03*, Apr. 2003.

A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage

James S. Plank
University of Tennessee
plank@cs.utk.edu

Jianqiang Luo
Wayne State University

Catherine D. Schuman
University of Tennessee

Lihao Xu
Wayne State University

Zooko Wilcox-O'Hearn
AllMyData, Inc.

Abstract

Over the past five years, large-scale storage installations have required fault-protection beyond RAID-5, leading to a flurry of research on and development of erasure codes for multiple disk failures. Numerous open-source implementations of various coding techniques are available to the general public. In this paper, we perform a head-to-head comparison of these implementations in encoding and decoding scenarios. Our goals are to compare codes and implementations, to discern whether theory matches practice, and to demonstrate how parameter selection, especially as it concerns memory, has a significant impact on a code's performance. Additional benefits are to give storage system designers an idea of what to expect in terms of coding performance when designing their storage systems, and to identify the places where further erasure coding research can have the most impact.

1 Introduction

In recent years, erasure codes have moved to the fore to prevent data loss in storage systems composed of multiple disks. Storage companies such as Allmydata [1], Cleversafe [7], Data Domain [36], Network Appliance [22] and Panasas [32] are all delivering products that use erasure codes beyond RAID-5 for data availability. Academic projects such as LoCI [3], Oceanstore [29], and Pergamum [31] are doing the same. And of equal importance, major technology corporations such as Hewlett Packard [34], IBM [12, 13] and Microsoft [15, 16] are performing active research on erasure codes for storage systems.

Along with proprietary implementations of erasure codes, there have been numerous open source implementations of a variety of erasure codes that are available for download [7, 19, 23, 26, 33]. The intent of most of these projects is to provide storage system developers

with high quality tools. As such, there is a need to understand how these codes and implementations perform.

In this paper, we compare the encoding and decoding performance of five open-source implementations of five different types of erasure codes: Classic Reed-Solomon codes [28], Cauchy Reed-Solomon codes [6], EVEN-ODD [4], Row Diagonal Parity (RDP) [8] and Minimal Density RAID-6 codes [5, 24, 25]. The latter three codes are specific to RAID-6 systems that can tolerate exactly two failures. Our exploration seeks not only to compare codes, but also to understand which features and parameters lead to good coding performance.

We summarize the main results as follows:

- The special-purpose RAID-6 codes vastly outperform their general-purpose counterparts. RDP performs the best of these by a narrow margin.
- Cauchy Reed-Solomon coding outperforms classic Reed-Solomon coding significantly, as long as attention is paid to generating good encoding matrices.
- An optimization called *Code-Specific Hybrid Reconstruction* [14] is necessary to achieve good decoding speeds in many of the codes.
- Parameter selection can have a huge impact on how well an implementation performs. Not only must the number of computational operations be considered, but also how the code interacts with the memory hierarchy, especially the caches.
- There is a need to achieve the levels of improvement that the RAID-6 codes show for higher numbers of failures.

Of the five libraries tested, *Zfec* [33] implemented the fastest classic Reed-Solomon coding, and *Jerasure* [26] implemented the fastest versions of the others.

2 Nomenclature and Erasure Codes

It is an unfortunate consequence of the history of erasure coding research that there is no unified nomenclature for erasure coding. We borrow terminology mostly from Hafner *et al* [14], but try to conform to more classic coding terminology (e.g. [5, 21]) when appropriate.

Our storage system is composed of an **array** of n disks, each of which is the same size. Of these n disks, k of them hold **data** and the remaining m hold **coding** information, often termed **parity**, which is calculated from the data. We label the data disks D_0, \dots, D_{k-1} and the parity disks C_0, \dots, C_{m-1} . A typical system is pictured in Figure 1.

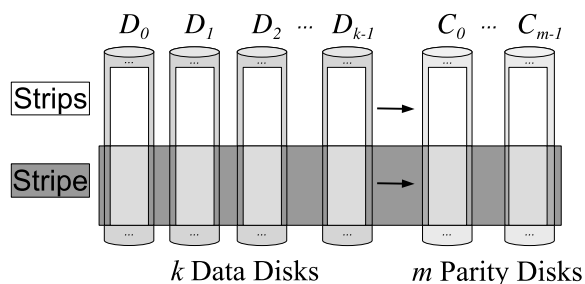


Figure 1: A typical storage system with erasure coding.

We are concerned with **Maximum Distance Separable (MDS)** codes, which have the property that if any m disks fail, the original data may be reconstructed [21]. When encoding, one partitions each disk into **strips** of a fixed size. Each parity strip is encoded using one strip from each data disk, and the collection of $k + m$ strips that encode together is called a **stripe**. Thus, as in Figure 1, one may view each disk as a collection of strips, and one may view the entire system as a collection of stripes. Stripes are each encoded independently, and therefore if one desires to rotate the data and parity among the n disks for load balancing, one may do so by switching the disks' identities for each stripe.

2.1 Reed-Solomon (RS) Codes

Reed-Solomon codes [28] have the longest history. The strip unit is a w -bit word, where w must be large enough that $n \leq 2^w + 1$. So that words may be manipulated efficiently, w is typically constrained so that words fall on machine word boundaries: $w \in \{8, 16, 32, 64\}$. However, as long as $n \leq 2^w + 1$, the value of w may be chosen at the discretion of the user. Most implementations choose $w = 8$, since their systems contain fewer than 256 disks, and $w = 8$ performs the best. Reed-Solomon codes treat each word as a number between 0 and $2^w - 1$, and operate on these numbers with *Galois*

Field arithmetic ($GF(2^w)$), which defines addition, multiplication and division on these words such that the system is closed and well-behaved [21].

The act of encoding with Reed-Solomon codes is simple linear algebra. A *Generator Matrix* is constructed from a *Vandermonde* matrix, and this matrix is multiplied by the k data words to create a *codeword* composed of the k data and m coding words. We picture the process in Figure 2 (note, we draw the transpose of the Generator Matrix to make the picture clearer).

$$\begin{array}{c}
 \begin{array}{|c|c|c|c|} \hline 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \\ \hline x_{00} & x_{01} & x_{02} & x_{03} \\ \hline x_{10} & x_{11} & x_{12} & x_{13} \\ \hline \end{array} \\
 G^T
 \end{array}
 *
 \begin{array}{c}
 \begin{array}{|c|} \hline d_0 \\ \hline d_1 \\ \hline d_2 \\ \hline d_3 \\ \hline \end{array} \\
 \text{Data}
 \end{array}
 =
 \begin{array}{c}
 \begin{array}{|c|} \hline d_0 \\ \hline d_1 \\ \hline d_2 \\ \hline d_3 \\ \hline c_0 \\ \hline c_1 \\ \hline \end{array} \\
 \text{Codeword}
 \end{array}
 \begin{array}{l}
 \rightarrow \text{Data} \\
 \rightarrow \text{Parity}
 \end{array}$$

Figure 2: Reed-Solomon coding for $k = 4$ and $m = 2$. Each element is a number between 0 and $2^w - 1$.

When disks fail, one decodes by deleting rows of G^T , inverting it, and multiplying the inverse by the surviving words. This process is equivalent to solving a set of independent linear equations. The construction of G^T from the Vandermonde matrix ensures that the matrix inversion is always successful.

In $GF(2^w)$, addition is equivalent to bitwise exclusive-or (XOR), and multiplication is more complex, typically implemented with multiplication tables or discrete logarithm tables [11]. For this reason, Reed-Solomon codes are considered expensive. There are several open-source implementations of RS coding, which we detail in Section 3.

2.2 Cauchy Reed-Solomon (CRS) Codes

CRS codes [6] modify RS codes in two ways. First, they employ a different construction of the Generator matrix using Cauchy matrices instead of Vandermonde matrices. Second, they eliminate the expensive multiplications of RS codes by converting them to extra XOR operations. Note, this second modification can apply to Vandermonde-based RS codes as well. This modification transforms G^T from a $n \times k$ matrix of w -bit words to a $wn \times wk$ matrix of bits. As with RS coding, w must be selected so that $n \leq 2^w + 1$.

Instead of operating on single words, CRS coding operates on entire strips. In particular, strips are partitioned into w *packets*, and these packets may be large. The act of coding now involves only XOR operations – a coding packet is constructed as the XOR of all data packets that

have a one bit in the coding packet's row of G^T . The process is depicted in Figure 3, which illustrates how the last coding packet is created as the XOR of the six data packets identified by the last row of G^T .

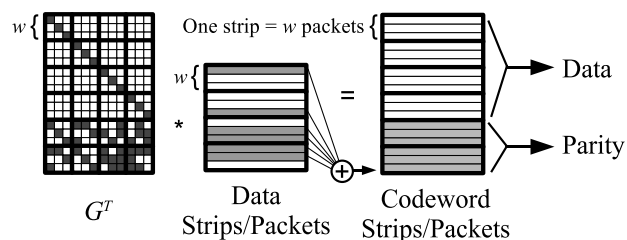


Figure 3: CRS example for $k = 4$ and $m = 2$.

To make XORs efficient, the packet size must be a multiple of the machine's word size. The strip size is therefore equal to w times the packet size. Since w no longer relates to the machine word sizes, w is not constrained to $[8, 16, 32, 64]$; instead, any value of w may be selected as long as $n \leq 2^w$.

Decoding in CRS is analogous to RS coding — all rows of G^T corresponding to failed packets are deleted, and the matrix is inverted and employed to recalculate the lost data.

Since the performance of CRS coding is directly related to the number of ones in G^T , there has been research on constructing Cauchy matrices that have fewer ones than the original CRS constructions [27]. The *Jerasure* library [26] uses additional matrix transformations to improve these matrices further. Additionally, in the restricted case when $m = 2$, the *Jerasure* library uses results of a previous enumeration of all Cauchy matrices to employ provably optimal matrices for all $w \leq 32$ [26].

2.3 EVENODD and RDP

EVENODD [4] and RDP [8] are two codes developed for the special case of RAID-6, which is when $m = 2$. Conventionally in RAID-6, the first parity drive is labeled P , and the second is labeled Q . The P drive is equivalent to the parity drive in a RAID-4 system, and the Q drive is defined by parity equations that have distinct patterns.

Although their original specifications use different terms, EVENODD and RDP fit the same paradigm as CRS coding, with strips being composed of w packets. In EVENODD, w is constrained such that $k + 1 \leq w$ and $w + 1$ is a prime number. In RDP, $w + 1$ must be prime and $k \leq w$. Both codes perform the best when $(w - k)$ is minimized. In particular, RDP achieves optimal encoding and decoding performance of $(k - 1)$ XOR operations per coding word when $k = w$ or $k + 1 = w$. Both codes' performance decreases as $(w - k)$ increases.

2.4 Minimal Density RAID-6 Codes

If we encode using a Generator bit-matrix for RAID-6, the matrix is quite constrained. In particular, the first kw rows of G^T compose an identity matrix, and in order for the P drive to be straight parity, the next w rows must contain k identity matrices. The only flexibility in a RAID-6 specification is the composition of the last w rows. In [5], Blaum and Roth demonstrate that when $k \leq w$, these remaining w rows must have at least $kw + k - 1$ ones for the code to be MDS. We term MDS matrices that achieve this lower bound *Minimal Density* codes.

There are three different constructions of Minimal Density codes for different values of w :

- **Blaum-Roth** codes when $w + 1$ is prime [5].
- **Liberation** codes when w is prime [25].
- The **Liber8tion** code when $w = 8$ [24].

These codes share the same performance characteristics. They encode with $(k - 1) + \frac{k-1}{2w}$ XOR operations per coding word. Thus, they perform better when w is large, achieving asymptotic optimality as $w \rightarrow \infty$. Their decoding performance is slightly worse, and requires a technique called *Code-Specific Hybrid Reconstruction* [14] to achieve near-optimal performance [25].

The Minimal Density codes also achieve near-optimal *updating* performance when individual pieces of data are modified [27]. This performance is significantly better than EVENODD and RDP, which are worse by a factor of roughly 1.5 [25].

2.5 Anvin's RAID-6 Optimization

In 2007, Anvin posted an optimization of RS encoding for RAID-6 [2]. For this optimization, the row of G^T corresponding to the P drive contains all ones, so that the P drive may be parity. The row corresponding to the Q drive contains the number 2^i in $GF(2^w)$ in column i (zero-indexed), so that the contents of the Q drive may be calculated by successively XOR-ing drive i 's data into the Q drive and multiplying that sum by two. Since multiplication by two may be implemented much faster than general multiplication in $GF(2^w)$, this optimizes the performance of encoding over standard RS implementations. Decoding remains unoptimized.

3 Open Source Libraries

We test five open source erasure coding libraries. These are all freely available libraries from various sources on the Internet, and range from brief proofs of concept

(e.g. *Luby*) to tuned and supported code intended for use in real systems (e.g. *Zfec*). We also tried the **Schifra** open source library [23], which is free but without documentation. We were unable to implement an encoder and decoder to perform a satisfactory comparison with the others. We present them chronologically.

Luby: CRS coding was developed at the ICSI lab in Berkeley, CA in the mid 1990's [6]. The authors released a C version of their codes in 1997, which is available from ICSI's web site [19]. The library supports all settings of k , m , w and packet sizes. The matrices employ the original constructions from [6], which are not optimized to minimize the number of ones.

Zfec: The *Zfec* library for erasure coding has been in development since 2007, but its roots have been around for over a decade. *Zfec* is built on top of a RS coding library developed for reliable multicast by Rizzo [30]. That library was based on previous work by Karn *et al* [18], and has seen wide use and tuning. *Zfec* is based on Vandermonde matrices when $w = 8$. The latest version (1.4.0) was posted in January, 2008 [33]. The library is programmable, portable and actively supported by the author. It includes command-line tools and APIs in C, Python and Haskell.

Jerasure: Jerasure is a C library released in 2007 that supports a wide variety of erasure codes, including RS coding, CRS coding, general Generator matrix and bit-matrix coding, and Minimal Density RAID-6 coding [26]. RS coding may be based on Vandermonde or Cauchy matrices, and w may be 8, 16 or 32. Anvin's optimization is included for RAID-6 applications. For CRS coding, *Jerasure* employs provably optimal encoding matrices for RAID-6, and constructs optimized matrices for larger values of m . Additionally, the three Minimal Density RAID-6 codes are supported. To improve performance of the bit-matrix codes, especially the decoding performance, the Code-Specific Hybrid Reconstruction optimization [14] is included. *Jerasure* is released under the GNU LGPL.

Cleversafe: In May, 2008, Cleversafe exported the first open source version of its dispersed storage system [7]. Written entirely in Java, it supports the same API as Cleversafe's proprietary system, which is notable as one of the first commercial distributed storage systems to implement availability beyond RAID-6. For this paper, we obtained a version containing just the the erasure coding part of the open source distribution. It is based on Luby's original CRS implementation [19] with $w = 8$.

EVENODD/RDP: There are no open source versions of EVENODD or RDP coding. However, RDP may be implemented as a bit-matrix code, which, when combined with Code-Specific Hybrid Reconstruction yields the same performance as the original specification of the code [16]. EVENODD may also be implemented with a

bit-matrix whose operations may be scheduled to achieve the code's original performance [16]. We use these observations to implement both codes as bit-matrices with tuned schedules in *Jerasure*. Since EVENODD and RDP codes are patented, this implementation is not available to the public, as its sole intent is for performance comparison.

4 Encoding Experiment

We perform two sets of experiments – one for encoding and one for decoding. For the encoding experiment, we seek to measure the performance of taking a large data file and splitting and encoding it into $n = k + m$ pieces, each of which will reside on a different disk, making the system tolerant to up to m disk failures. Our encoder thus reads a data file, encodes it, and writes it to $k + m$ data/coding files, measuring the performance of the encoding operations.

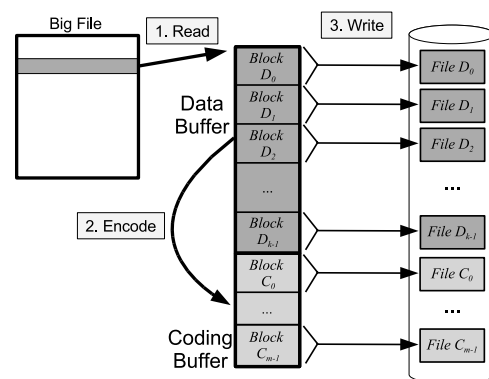


Figure 4: The encoder utilizes a data buffer and a coding buffer to encode a large file in stages.

Since memory utilization is a concern, and since large files exceed the capacity of most computers' memories, our encoder employs two fixed-size buffers, a *Data Buffer* partitioned into k blocks and a *Coding Buffer* partitioned into m blocks. The encoder reads an entire data buffer's worth of data from the big file, encodes it into the coding buffer and then writes the contents of both buffers to $k + m$ separate files. It repeats this process until the file is totally encoded, recording both the total time and the encoding time. The high level process is pictured in Figure 4.

The blocks of the buffer are each partitioned into s strips, and each strip is partitioned either into words of size w (RS coding, where $w \in \{8, 16, 32, 64\}$), or into w packets of a fixed size **PS** (all other codes – recall Figure 3). To be specific, each block D_i (and C_j) is partitioned into strips $DS_{i,0}, \dots, DS_{i,s-1}$.

(and $CS_{j,0}, \dots, CS_{j,s-1}$), each of size $w\mathbf{PS}$. Thus, the data and coding buffer sizes are dependent on the various parameters. Specifically, the data buffer size equals $(ksw\mathbf{PS})$ and the coding buffer size equals $(msw\mathbf{PS})$.

Encoding is done on a stripe-by-stripe basis. First, the data strips $DS_{0,0}, \dots, DS_{k-1,0}$ are encoded into the coding strips $CS_{0,0}, \dots, CS_{m-1,0}$. This completes the encoding of stripe 0, pictured in Figure 5. Each of the s stripes is successively encoded in this manner.

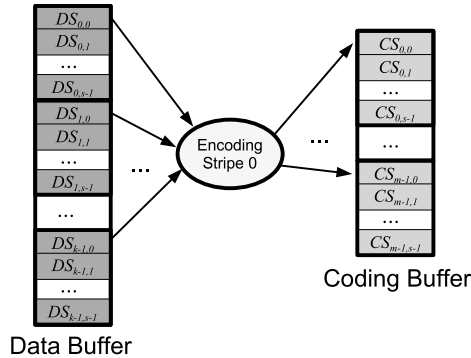


Figure 5: How the data and coding buffers are partitioned, and the encoding of Stripe 0 from data strips $DS_{0,0}, \dots, DS_{k-1,0}$ into coding strips $CS_{0,0}, \dots, CS_{m-1,0}$.

Thus, there are multiple parameters that the encoder allows the user to set. These are k , m , w (subject to the code's constraints), s and \mathbf{PS} . When we mention setting the buffer size below, we are referring to the size of the data buffer, which is $(ksw\mathbf{PS})$.

4.1 Machines for Experimentation

We employed two machines for experimentation. Neither is exceptionally high-end, but each represents middle-range commodity processors, which should be able to encode and decode comfortably within the I/O speed limits of the fastest disks. The first is a Macbook with a 32-bit 2GHz Intel Core Duo processor, with 1GB of RAM, a L1 cache of 32KB and a L2 cache of 2MB. Although the machine has two cores, the encoder only utilizes one. The operating system is Mac OS X, version 10.4.11, and the encoder is executed in user space while no other user programs are being executed. As a baseline, we recorded a **memcpy()** speed of 6.13 GB/sec and an **XOR** speed of 2.43 GB/sec.

The second machine is a Dell workstation with an Intel Pentium 4 CPU running at 1.5GHz with 1GB of RAM, an 8KB L1 cache and a 256KB L2 cache. The operating system is Debian GNU Linux revision 2.6.8-2-686, and

the machine is a 32-bit machine. The **memcpy()** speed is 2.92 GB/sec and the **XOR** speed is 1.32 GB/sec.

4.2 Encoding with Large Files

Our intent was to measure the actual performance of encoding a large video file. However, doing large amounts of I/O causes a great deal of variability in performance timings. We exemplify with Figure 6. The data is from the Macbook, where we use a 256 MB video file for input. The encoder works as described in Section 4 with $k = 10$ and $m = 6$. However, we perform no real encoding. Instead we simply zero the bytes of the coding buffer before writing it to disk. In the figure, we modify the size of the data buffer from a small size of 64 KB to 256 MB – the size of the video file itself.

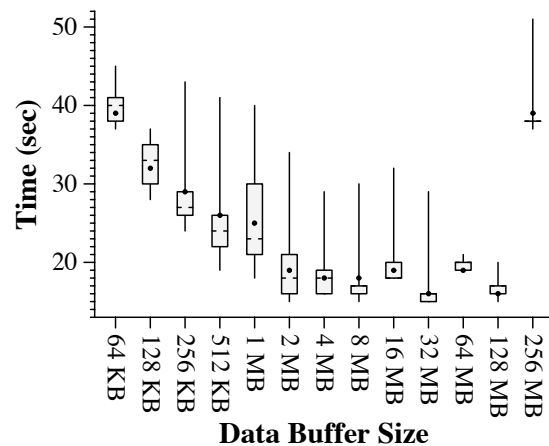


Figure 6: Times to read a 256 MB video, perform a dummy encoding when $k = 10$ and $m = 6$, and write to 16 data/coding files.

In Figure 6, each data point is the result of ten runs executed in random order. A tukey plot is given, which has bars to the maximum and minimum values, a box encompassing the first to the third quartile, hash marks at the median and a dot at the mean. While there is a clear trend toward improving performance as the data buffer grows to 128 MB, the variability in performance is colossal: between 15 and 20 seconds for many runs. Running Unix's **split** utility on the file reveals similar variability.

Because of this variability, the tests that follow remove the I/O from the encoder. Instead, we simulate reading by filling the buffer with random bytes, and we simulate writing by zeroing the buffers. This reduces the variability of the runs tremendously – the results that follow are all averages of over 10 runs, whose maximum and minimum values differ by less than 0.5 percent. The encoder measures the times of all coding activities using Unix's **gettimeofday()**. To confirm that these times

are accurate, we also subtracted the wall clock time of a dummy control from the wall clock time of the encoder, and the two matched to within one percent.

Figure 6 suggests that the size of the data buffer can impact performance, although it is unclear whether the impact comes from memory effects or from the file system. To explore this, we performed a second set of tests that modify the size of the data buffer while performing a dummy encoding. We do not graph the results, but they show that with the I/O removed, the effects of modifying the buffer size are negligible. Thus, in the results that follow, we maintain a data buffer size of roughly 100 KB. Since actual buffer sizes depend on k , m , w and **PS**, they cannot be affixed to a constant value; instead, they are chosen to be in the ballpark of 100 KB. This is large enough to support efficient I/O, but not so large that it consumes all of a machine's memory, since in real systems the processors may be multitasking.

4.3 Parameter Space

We test four combinations of k and m – we will denote them by $[k, m]$. Two combinations are RAID-6 scenarios: $[6, 2]$ and $[14, 2]$. The other two represent 16-disk stripes with more fault-tolerance: $[12, 4]$ and $[10, 6]$. We chose these combinations because they represent values that are likely to be seen in actual usage. Although large and wide-area storage installations are composed of much larger numbers of disks, the stripe sizes tend to stay within this medium range, because the benefits of large stripe sizes show diminishing returns compared to the penalty of extra coding overhead in terms of encoding performance and memory use. For example, Cleversafe's widely dispersed storage system uses $[10, 6]$ as its default [7]; Allmydata's archival online backup system uses $[3, 7]$, and both Panasas [32] and Pergamum [31] report keeping their stripe sizes at or under 16.

For each code and implementation, we test its performance by encoding a randomly generated file that is 1 GB in size. We test all legal values of $w \leq 32$. This results in the following tests.

- *Zfec*: RS coding, $w = 8$ for all combinations of $[k, m]$.
- *Luby*: CRS coding, $w \in \{4, \dots, 12\}$ for all combinations of $[k, m]$, and $w = 3$ for $[6, 2]$.
- *Cleversafe*: CRS coding, $w = 8$ for all combinations of $[k, m]$.
- *Jerasure*:
 - RS coding, $w \in \{8, 16, 32\}$ for all combinations of $[k, m]$. Anvin's optimization is included for the RAID-6 tests.

- CRS coding, $w \in \{4, \dots, 32\}$ for all combinations of $[k, m]$, and $w = 3$ for $[6, 2]$.
- Blaum-Roth codes, $w \in \{6, 10, 12\}$ for $[6, 2]$ and $w \in \{16, 18, 22, 28, 30\}$ for $[6, 2]$ and $[14, 2]$.
- Liberation codes, $w \in \{7, 11, 13\}$ for $[6, 2]$ and $w \in \{17, 19, 23, 29, 31\}$ for $[6, 2]$ and $[14, 2]$.
- The Liber8tion code, $w = 8$ for $[6, 2]$.

- *EVENODD*: Same parameters as Blaum-Roth codes in *Jerasure* above.

- *RDP*: Same parameters as *EVENODD*.

4.4 Impact of the Packet Size

Our experience with erasure coding led us to experiment first with modifying the packet sizes of the encoder. There is a clear tradeoff: lower packet sizes have less tight XOR loops, but better cache behavior. Higher packet sizes perform XORs over larger regions, but cause more cache misses. To exemplify, consider Figure 7, which shows the performance of RDP on the $[6, 2]$ configuration when $w = 6$, on the Macbook. We test every packet size from 4 to 10000 and display the speed of encoding.

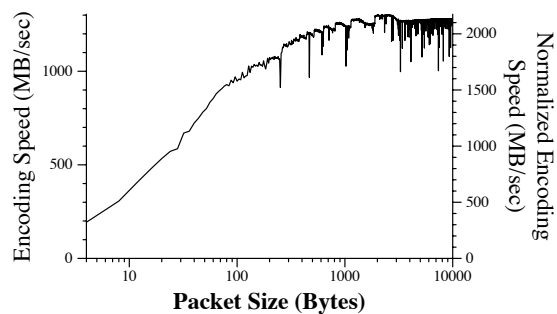


Figure 7: The effect of modifying the packet size on RDP coding, $k = 6$, $m = 2$, $w = 6$ on the Macbook.

We display two y-axes. On the left is the encoding speed. This is the size of the input file divided by the time spent encoding and is the most natural metric to plot. On the right, we normalize the encoding speed so that we may compare the performance of encoding across configurations. The normalized encoding speed is calculated as:

$$\frac{(\text{Encoding Speed}) m(k-1)}{k}. \quad (1)$$

This is derived as follows. Let S be the file's size and t be the time to encode. The file is split and encoded

into $m + k$ files, each of size $\frac{S}{k}$. The encoding process itself creates $\frac{Sm}{k}$ bytes worth of coding data, and therefore the speed per coding byte is $\frac{Sm}{kt}$. Optimal encoding takes $k - 1$ XOR operations per coding drive [35]; therefore we can normalize the speed by dividing the time by $k - 1$, leaving us with $\frac{Sm(k-1)}{kt}$, or Equation 1 for the normalized encoding speed.

The shape of this curve is typical for all codes on both machines. In general, higher packet sizes perform better than lower ones; however there is a maximum performance point which is achieved when the code makes best use of the L1 cache. In this test, the optimal packet size is 2400 bytes, achieving a normalized encoding speed of 2172 MB/sec. Unfortunately, this curve does not monotonically increase to nor decrease from its optimal value. Worse, there can be radical dips in performance between adjacent packet sizes, due to collisions between cache entries. For example, at packet sizes 7732, 7736 and 7740, the normalized encoding speeds are 2133, 2066 and 2129 MB/sec, respectively. We reiterate that each data point in our graphs represents over 10 runs, and the repetitions are consistent to within 0.5 percent.

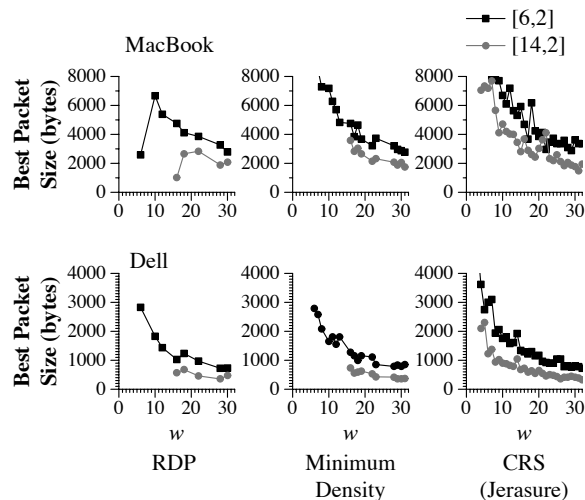


Figure 8: The effect of modifying w on the best packet sizes found.

We do not attempt to find the optimal packet sizes for each of the codes. Instead, we perform a search algorithm that works as follows. We test a region r of packet sizes by testing each packet size from r to $r + 36$ (packet sizes must be a multiple of 4). We set the region's performance to be the average of the five best tests. To start our search, we test all regions that are powers of two from 64 to 32K. We then iterate, finding the best region r , and then testing the two regions that are halfway between the two values of r that we have tested that are adjacent to r . We do this until there are no more regions to test, and se-

lect the packet size of all tested that performed the best. For example, the search for the RDP instance of Figure 7 tested only 202 packet sizes (as opposed to 2500 to generate Figure 7) to arrive at a packet size of 2588 bytes, which encodes at a normalized speed of 2164 MB/sec (0.3% worse than the best packet size of 2400 bytes).

One expects the optimal packet size to decrease as k , m and w increase, because each of these increases the stripe size. Thus smaller packets are necessary for most of the stripe to fit into cache. We explore this effect in Figure 8, where we show the best packet sizes found for different sets of codes – RDP, Minimum Density, and Jerasure's CRS – in the two RAID-6 configurations. For each code, the larger value of k results in a smaller packet size, and as a rough trend, as w increases, the best packet size decreases.

4.5 Overall Encoding Performance

We now present the performance of each of the codes and implementations. In the codes that allow a packet size to be set, we select the best packet size from the above search. The results for the [6,2] configuration are in Figure 9.

Although the graphs for both machines appear similar, there are interesting features of both. We concentrate first on the MacBook. The specialized RAID-6 codes outperform all others, with RDP's performance with $w = 6$ performing the best. This result is expected, as RDP achieves optimal performance when $k = w$.

The performance of these codes is typically quantified by the number of XOR operations performed [5, 4, 8, 25, 24]. To measure how well number of XORs matches actual performance, we present the number of gigabytes XOR'd by each code in Figure 10.

On the MacBook, the number of XORs is an excellent indicator of performance, with a few exceptions (CRS codes for $w \in \{21, 22, 32\}$). As predicted by XOR count, RDP's performance suffers as w increases, while the Minimal Density codes show better performance. Of the three special-purpose RAID-6 codes, EVENODD performs the worst, although the margins are not large (the worst performing EVENODD encodes at 89% of the speed of the best RDP).

The performance of Jerasure's implementation the CRS codes is also excellent, although the choice of w is very important. The number of ones in the CRS generator matrices depends on the number of bits in the Galois Field's primitive polynomial. The polynomials for $w \in \{8, 12, 13, 14, 16, 19, 24, 26, 27, 30, 32\}$ have one more bit than the others, resulting in worse performance. This is important, as $w \in \{8, 16, 32\}$ are very natural choices since they allow strip sizes to be powers of two.

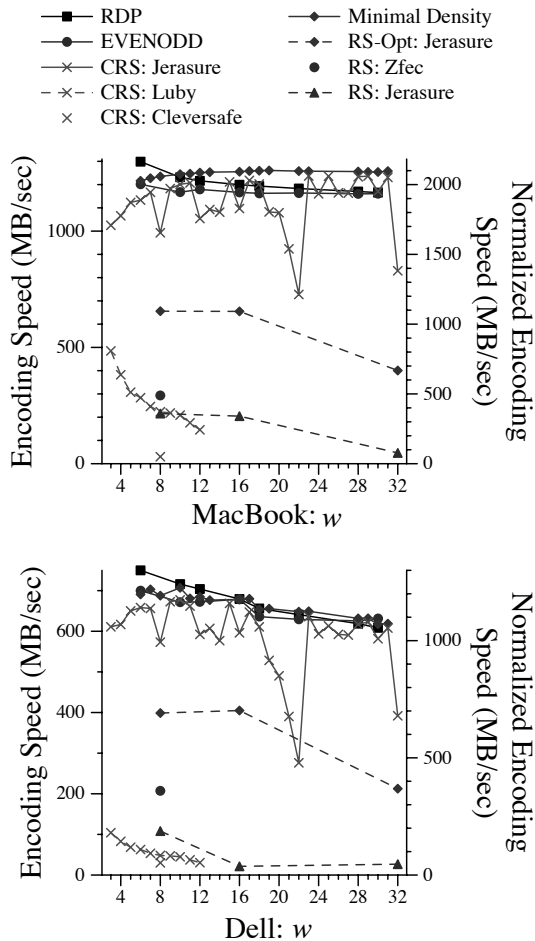


Figure 9: Encoding performance for [6,2].

Returning back to figure 9, the *Luby* and *Cleversafe* implementations of CRS coding perform much worse than *Jerasure*. There are several reasons for this. First, they do not optimize the generator matrix in terms of number of ones, and thus perform many more XOR operations, from 3.2 GB of XORs when $w = 3$ to 13.5 GB when $w = 12$. Second, both codes use a dense, bit-packed representation of the generator matrix, which means that they spend quite a bit of time performing bit operations to check matrix entries, many of which are zeros and could be omitted. *Jerasure* converts the matrix to a schedule which eliminates all of the matrix traversal and entry checking during encoding. *Cleversafe*'s poor performance relative to *Luby* can most likely be attributed to the Java implementation and the fact that the packet size is hard coded to be very small (since *Cleversafe* routinely distributes strips in units of 1K).

Of the RS implementations, the implementation tailored for RAID-6 (labeled "RS-Opt") performs at a much higher rate than the others. This is due to the fact that

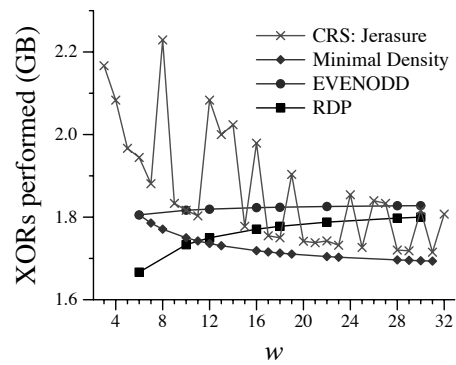


Figure 10: Gigabytes XOR'd by each code in the [6,2] tests. The number of XORs is independent of the machine used.

it does not perform general-purpose Galois Field multiplication over w -bit words, but instead performs a machine word's worth of multiplication by two at a time. Its performance is better when $w \leq 16$, which is not a limitation as $w = 16$ can handle a system with a total of 64K drives. The *Zfec* implementation of RS coding outperforms the others. This is due to the heavily tuned implementation, which performs explicit loop unrolling and hard-wires many features of $GF(2^8)$ which the other libraries do not. Both *Zfec* and *Jerasure* use pre-computed multiplication and division tables for $GF(2^8)$. For $w = 16$, *Jerasure* uses discrete logarithms, and for $w = 32$, it uses a recursive table-lookup scheme. Additional implementation options for the underlying Galois Field arithmetic are discussed in [11].

The results on the Dell are similar to the MacBook with some significant differences. The first is that larger values of w perform worse relative to smaller values, regardless of their XOR counts. While the Minimum Density codes eventually outperform RDP for larger w , their overall performance is far worse than the best performing RDP instance. For example, Liberation's encoding speed when $w = 31$ is 82% of RDP's speed when $w = 6$, as opposed to 97% on the MacBook. We suspect that the reason for this is the smaller L1 cache on the Dell, which penalizes the strip sizes of the larger w .

The final difference between the MacBook and the Dell is that *Jerasure*'s RS performance for $w = 16$ is much worse than for $w = 8$. We suspect that this is because *Jerasure*'s logarithm tables are not optimized for space, consuming 1.5 MB of memory, since there are six tables of 256 KB each [26]. The lower bound is two 128 KB tables, which should exhibit better behavior on the Dell's limited cache.

Figure 11 displays the results for [14,2] (we omit *Cleversafe* since its performance is so much worse than the others). The trends are similar to [6,2], with the ex-

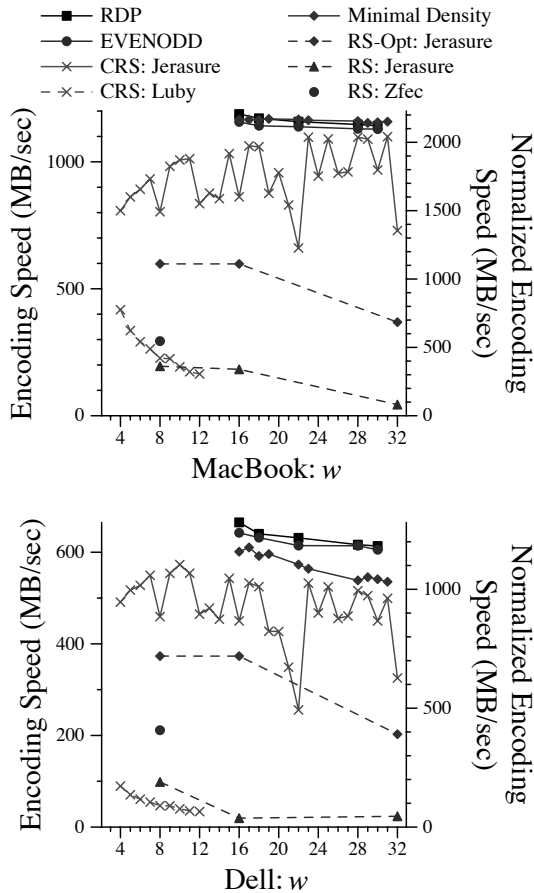


Figure 11: Encoding performance for [14,2].

ception that on the Dell, the Minimum Density codes perform significantly worse than RDP and EVENODD, even though their XOR counts follow the performance of the MacBook. The definition of the normalized encoding speed means that if a code is encoding optimally, its normalized encoding speed should match the XOR speed. In both machines, RDP's [14,2] normalized encoding speed comes closest to the measured XOR speed, meaning that in implementation as in theory, this is an extremely efficient code.

Figure 12 displays the results for [12,4]. Since this is no longer a RAID-6 scenario, only the RS and CRS codes are displayed. The normalized performance of *Jerasure*'s CRS coding is much worse now because the generator matrices are more dense and cannot be optimized as they can when $m = 2$. As such, the codes perform more XOR operations than when $k = 14$. For example, when $w = 4$ *Jerasure*'s CRS implementation performs 17.88 XORs per coding word; optimal is 11. This is why the normalized coding speed is much slower than in the best RAID-6 cases. Since *Luby*'s code does not optimize the generator matrix, it performs more XORs

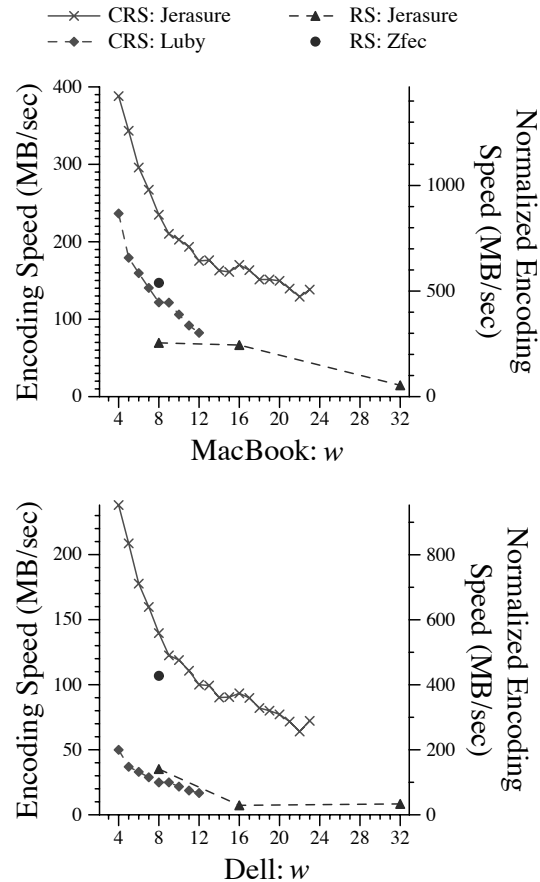


Figure 12: Encoding performance for [12,4].

(23.5 per word, as opposed to 17.88 for *Jerasure*), and as a result is slower.

The RS codes show the same performance as in the other tests. In particular, *Zfec*'s normalized performance is roughly the same in all cases. For space purposes, we omit the [10,6] results as they show the same trends as the [12,4] case. The peak performer is *Jerasure*'s CRS, achieving a normalized speed of 1409 MB/sec on the MacBook and 869.4 MB/sec on the Dell. *Zfec*'s normalized encoding speeds are similar to the others: 528.4 MB/sec on the MacBook and 380.2 MB/sec on the Dell.

5 Decoding Performance

To test the performance of decoding, we converted the encoder program to perform decoding as well. Specifically, the decoder chooses m random data drives, and then after each encoding iteration, it zeros the buffers for those drives and decodes. We only decode data drives for two reasons. First, it represents the hardest decoding case, since all of the coding information must be used. Second, all of the libraries except *Jerasure* decode only

the data, and do not allow for individual coding strips to be re-encoded without re-encoding all of them. While we could have modified those libraries to re-encode individually, we did not feel that it was in the spirit of the evaluation. Before testing, we wrote code to double-check that the erased data was decoded correctly, and in all cases it was.

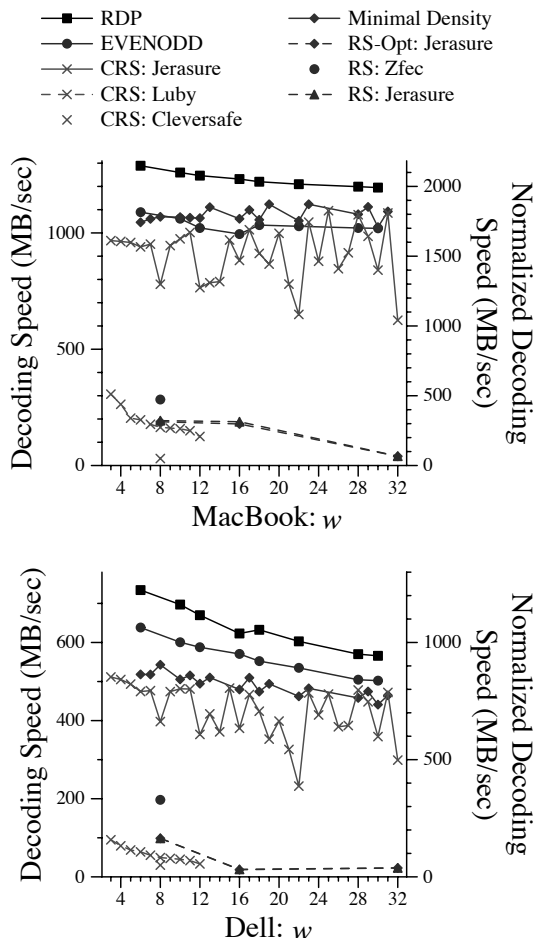


Figure 13: Decoding performance for [6,2].

We show the performance of two configurations: [6,2] in Figure 13 and [12,4] in Figure 14. The results are best viewed in comparison to Figures 9 and 12. The results on the MacBook tend to match theory. RDP decodes as it encodes, and the two sets of speeds match very closely. EVENODD and the Minimal Density codes both have slightly more complexity in decoding, which is reflected in the graph. As mentioned in [24], the Minimal Density codes benefit greatly from Code-Specific Hybrid Reconstruction [14], which is implemented in *Jersure*. Without the optimization, the decoding performance of these codes would be unacceptable. For example, in the [6,2] configuration on the MacBook, the Liberation code

for $w = 31$ decodes at a normalized rate of 1820 MB/sec. Without Code-Specific Hybrid Reconstruction, the rate is a factor of six slower: 302.7 MB/sec. CRS coding also benefits from the optimization. Again, using an example where $w = 31$, normalized speed with the optimization is 1809 MB/s, and without it is 261.5 MB/sec.

The RS decoders perform identically to their encoding counterparts with the exception of the RAID-6 optimized version. This is because the optimization applies only to encoding and defaults to standard RS decoding. Since the only difference between RS encoding and decoding is the inversion of a $k \times k$ matrix, the fact that encoding and decoding performance match is expected.

On the Dell, the trends between the various codes follow the encoding tests. In particular, larger values of w are penalized more by the small cache.

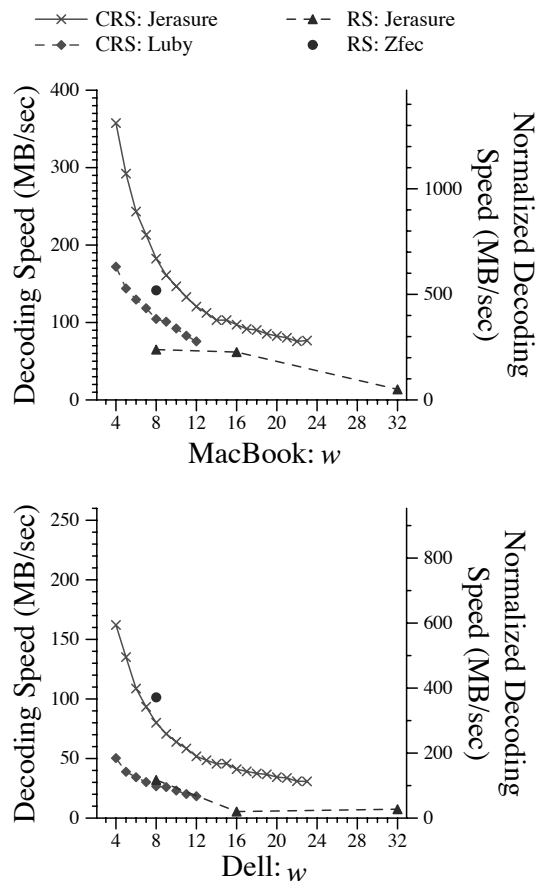


Figure 14: Decoding performance for [12,4].

In the [12,4] tests, the performance trends of the CRS codes are the same, although the decoding proceeds more slowly. This is more pronounced in *Jersure*'s implementation than in *Luby*'s, and can be explained by XORs. In *Jersure*, the program attempts to minimize the number of ones in the encoding matrix, without regard to the

decoding matrix. For example, when $w = 4$, CRS encoding requires 5.96 GB of XORs. In a decoding example, it requires 14.1 GB of XORs, and with Code-Specific Hybrid Reconstruction, that number is reduced to 12.6. *Luby's* implementation does not optimize the encoding matrix, and therefore the penalty of decoding is not as great.

As with the [6,2] tests, the performance of RS coding remains identical to decoding.

6 XOR Units

This section is somewhat obvious, but it does bear mentioning that the unit of XOR used by the encoding/decoding software should match the largest possible XOR unit of the machine. For example, on 32-bit machines like the MacBook and the Dell, the **long** and **int** types are both four bytes, while the **char** and **short** types are one and two bytes, respectively. On 64-bit machines, the **long** type is eight bytes. To illustrate the dramatic impact of word size selection for XOR operations, we display RDP performance for the [6,2] configuration ($w = 6$) on the two 32-bit machines and on an HP dc7600 workstation with a 64-bit Pentium D860 processor running at 2.8 GHz. The results in Figure 15 are expected.

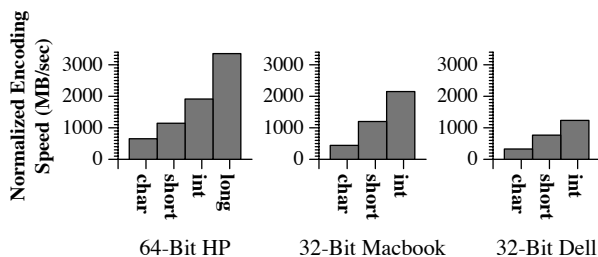


Figure 15: Effect of changing the XOR unit of RDP encoding when $w = 6$ in the [6,2] configuration.

The performance penalty at each successively smaller word size is roughly a factor of two, since twice as many XORs are being performed. All the libraries tested in this paper perform XORs with the widest word possible. This also displays how 64-bit are especially tailored for these types of operations.

7 Conclusions

Given the speeds of current disks, the libraries explored here perform at rates that are easily fast enough to build high performance, reliable storage systems. We offer the following lessons learned from our exploration and experimentation:

RAID-6: The three RAID-6 codes, plus *Jerasure's* implementation of CRS coding for RAID-6, all perform much faster than the general-purpose codes. Attention must be paid to the selection of w for these codes: for RDP and EVENODD, it should be as low as possible; for Minimal Density codes, it should be as high as the caching behavior allows, and for CRS, it should be selected so that the primitive polynomial has a minimal number of ones. Note that $w \in \{8, 16, 32\}$ are all bad for CRS coding. Anvin's optimization is a significant improvement of generalized RS coding, but does not attain the levels of the special-purpose codes.

CRS vs. RS: For non-RAID-6 applications, CRS coding performs much better than RS coding, but now w should be chosen to be as small as possible, and attention should be paid to reduce the number of ones in the generator matrix. Additionally, a dense matrix representation should not be used for the generator matrix while encoding and decoding.

Parameter Selection: In addition to w , the packet sizes of the codes should be chosen to yield good cache behavior. To achieve an ideal packet size, experimentation is important; although there is a balance point between too small and too large, some packet sizes perform poorly due to direct-mapped cache behavior, and therefore finding an ideal packet size takes more effort than executing a simple binary search. As reported by Greenan with respect to Galois Field arithmetic [11], architectural features and memory behavior interact in such a way that makes it hard to predict the optimal parameters for encoding operations. In this paper, we semi-automate it by using the region-based search of Section 4.4.

Minimizing the Cache/Memory Footprint: On some machines, the implementation must pay attention to memory and cache. For example, *Jerasure's* RS implementation performs poorly on the Dell when $w = 16$ because it is wasteful of memory, while on the MacBook its memory usage does not penalize as much. Part of *Zfec's* better performance comes from its smaller memory footprint. In a similar vein, we have seen improvements in the performance of the XOR codes by re-ordering the XOR operations to minimize cache replacements [20]. We anticipate further performance gains through this technique.

Beyond RAID-6: The place where future research will have the biggest impact is for larger values of m . The RAID-6 codes are extremely successful in delivering higher performance than their general-purpose counterparts. More research needs to be performed on special-purpose codes beyond RAID-6, and implementations need to take advantage of the special-purpose codes that already exist [9, 10, 17].

Multicore: As modern architectures shift more universally toward multicore, it will be an additional challenge for open source libraries to exploit the opportunities of multiple processors on a board. As demonstrated in this paper, attention to the processor/cache interaction will be paramount for high performance.

8 Acknowledgements

This material is based upon work supported by the National Science Foundation under grants CNS-0615221 and IIS-0541527. The authors are greatly indebted to Ilya Volvolski and Jason Resch from Cleversafe for providing us with the erasure coding core of their open source storage dispersal system. The authors also thank Hakim Weatherspoon for his helpful and detailed comments on the paper.

References

- [1] ALLMYDATA. Unlimited online backup, storage, and sharing. <http://allmydata.com>, 2008.
- [2] ANVIN, H. P. The mathematics of RAID-6. <http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf>, 2007.
- [3] BECK, M., ARNOLD, D., BASSI, A., BERMAN, F., CASANOVA, H., DONGARRA, J., MOORE, T., OBERTELLI, G., PLANK, J. S., SWANY, M., VADHIYAR, S., AND WOLSKI, R. Logistical computing and internetworking: Middleware for the use of storage in communication. In *Third Annual International Workshop on Active Middleware Services (AMS)* (San Francisco, August 2001).
- [4] BLAUM, M., BRADY, J., BRUCK, J., AND MENON, J. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computing* 44, 2 (February 1995), 192–202.
- [5] BLAUM, M., AND ROTH, R. M. On lowest density MDS codes. *IEEE Transactions on Information Theory* 45, 1 (January 1999), 46–59.
- [6] BLOMER, J., KALFANE, M., KARPINSKI, M., KARP, R., LUBY, M., AND ZUCKERMAN, D. An XOR-based erasure-resilient coding scheme. Tech. Rep. TR-95-048, International Computer Science Institute, August 1995.
- [7] CLEVERSAFE, INC. Cleversafe Dispersed Storage. Open source code distribution: <http://www.cleversafe.org/downloads>, 2008.
- [8] CORBETT, P., ENGLISH, B., GOEL, A., GRACANAC, T., KLEIMAN, S., LEONG, J., AND SANKAR, S. Row diagonal parity for double disk failure correction. In *3rd Usenix Conference on File and Storage Technologies* (San Francisco, CA, March 2004).
- [9] FENG, G., DENG, R., BAO, F., AND SHEN, J. New efficient MDS array codes for RAID Part I: Reed-Solomon-like codes for tolerating three disk failures. *IEEE Transactions on Computers* 54, 9 (September 2005), 1071–1080.
- [10] FENG, G., DENG, R., BAO, F., AND SHEN, J. New efficient MDS array codes for RAID Part II: Rabin-like codes for tolerating multiple (≥ 4) disk failures. *IEEE Transactions on Computers* 54, 12 (December 2005), 1473–1483.
- [11] GREENAN, K., MILLER, E., AND SCHWARTZ, T. J. Optimizing Galois Field arithmetic for diverse processor architectures and applications. In *MASCOTS 2008: 16th IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (Baltimore, MD, September 2008).
- [12] HAFNER, J. L. WEAVER Codes: Highly fault tolerant erasure codes for storage systems. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies* (San Francisco, December 2005), pp. 211–224.
- [13] HAFNER, J. L. HoVer erasure codes for disk arrays. In *DSN-2006: The International Conference on Dependable Systems and Networks* (Philadelphia, June 2006), IEEE.
- [14] HAFNER, J. L., DEENADHAYALAN, V., RAO, K. K., AND TOMLIN, A. Matrix methods for lost data reconstruction in erasure codes. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies* (San Francisco, December 2005), pp. 183–196.
- [15] HUANG, C., CHEN, M., AND LI, J. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. In *NCA-07: 6th IEEE International Symposium on Network Computing Applications* (Cambridge, MA, July 2007).
- [16] HUANG, C., LI, J., AND CHEN, M. On optimizing XOR-based codes for fault-tolerant storage applications. In *ITW'07, Information Theory Workshop* (Tahoe City, CA, September 2007), IEEE, pp. 218–223.

- [17] HUANG, C., AND XU, L. STAR: An efficient coding scheme for correcting triple storage node failures. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies* (San Francisco, December 2005), pp. 197–210.
- [18] KARN, P. Dsp and fec library. <http://www.ka9q.net/code/fec/>, 2007.
- [19] LUBY, M. Code for Cauchy Reed-Solomon coding. Uuencoded tar file: <http://www.icsi.berkeley.edu/~luby/cauchy.tar.uu>, 1997.
- [20] LUO, J., XU, L., AND PLANK, J. S. An efficient XOR-Scheduling algorithm for erasure code encoding. Tech. Rep. Computer Science, Wayne State University, December 2008.
- [21] MACWILLIAMS, F. J., AND SLOANE, N. J. A. *The Theory of Error-Correcting Codes, Part I*. North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.
- [22] NISBET, B. FAS storage systems: Laying the foundation for application availability. Network Appliance white paper: <http://www.netapp.com/us/library/analyst-reports/ar1056.html>, February 2008.
- [23] PARTOW, A. Schifra Reed-Solomon ECC Library. Open source code distribution: <http://www.schifra.com/downloads.html>, 2000-2007.
- [24] PLANK, J. S. A new minimum density RAID-6 code with a word size of eight. In *NCA-08: 7th IEEE International Symposium on Network Computing Applications* (Cambridge, MA, July 2008).
- [25] PLANK, J. S. The RAID-6 Liberation codes. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 97–110.
- [26] PLANK, J. S., SIMMERMAN, S., AND SCHUMAN, C. D. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Tech. Rep. CS-08-627, University of Tennessee, August 2008.
- [27] PLANK, J. S., AND XU, L. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *NCA-06: 5th IEEE International Symposium on Network Computing Applications* (Cambridge, MA, July 2006).
- [28] REED, I. S., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics* 8 (1960), 300–304.
- [29] RHEA, S., WELLS, C., EATON, P., GEELS, D., ZHAO, B., WEATHERSPOON, H., AND KUBIA-TOWICZ, J. Maintenance-free global data storage. *IEEE Internet Computing* 5, 5 (2001), 40–49.
- [30] RIZZO, L. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computer Communication Review* 27, 2 (1997), 24–36.
- [31] STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 1–16.
- [32] WELCH, B., UNANGST, M., ABBASI, Z., GIBSON, G., MUELLER, B., SMALL, J., ZELENKA, J., AND ZHOU, B. Scalable performance of the Panasas parallel file system. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 17–33.
- [33] WILCOX-O’HEARN, Z. Zfec 1.4.0. Open source code distribution: <http://pypi.python.org/pypi/zfec>, 2008.
- [34] WYLIE, J. J., AND SWAMINATHAN, R. Determining fault tolerance of XOR-based erasure codes efficiently. In *DSN-2007: The International Conference on Dependable Systems and Networks* (Edinburgh, Scotland, June 2007), IEEE.
- [35] XU, L., AND BRUCK, J. X-Code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory* 45, 1 (January 1999), 272–276.
- [36] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 269–282.

Tiered Fault Tolerance for Long-Term Integrity

Byung-Gon Chun, Petros Maniatis
Intel Research Berkeley

Scott Shenker, John Kubiatowicz
University of California at Berkeley

Abstract

Fault-tolerant services typically make assumptions about the type and maximum number of faults that they can tolerate while providing their correctness guarantees; when such a *fault threshold* is violated, correctness is lost. We revisit the notion of fault thresholds in the context of long-term archival storage. We observe that fault thresholds are inevitably violated in long-term services, making traditional fault tolerance inapplicable to the long-term. In this work, we undertake a “reallocation of the fault-tolerance budget” of a long-term service. We split the service into service pieces, each of which can tolerate a different number of faults without failing (and without causing the whole service to fail): each piece can be either in a critical *trusted fault tier*, which must never fail, or an *untrusted fault tier*, which can fail massively and often, or other fault tiers in between. By carefully engineering the split of a long-term service into pieces that must obey distinct fault thresholds, we can prolong its inevitable demise. We demonstrate this approach with *Bonafide*, a long-term key-value store that, unlike all similar systems proposed in the literature, maintains integrity in the face of Byzantine faults *without requiring self-certified data*. We describe the notion of tiered fault tolerance, the design, implementation, and experimental evaluation of *Bonafide*, and argue that our approach is a practical yet significant improvement over the state of the art for long-term services.

1 Introduction

Current fault-tolerant replicated service designs are often unsuitable for long-term applications, such as archival storage for digital artifacts, which is gaining importance for business [42], regulatory [5, 6], and cultural [36] reasons. This unsuitability results from the typical fault assumptions on which the correctness of such systems is conditioned. For example, in typical Byzantine-fault tolerant (BFT) replicated systems [13], it is assumed that the number of faulty replicas is always less than some fixed threshold such as $1/3$ of the replica population.

In typical, “near-term” applications, such a uniform-threshold-based fault assumption can be reasonable and achievable. For example, one can argue that in a well-maintained population of diverse, high-assurance replica servers, by the time a third of the population is broken into or just grows faulty, the operators of faulty replicas can repair them. Thus, the repair reduces the number of faulty replicas, averts a threshold breach, and thereby keeps the system’s fault assumption inviolate.

Unfortunately, this reasoning falls apart for applications and deployments with a long-term horizon, say many decades. Whereas a population of replica servers can be plausibly “well-maintained” enough for a few years, it is difficult to protect perfectly from momentary threshold breaches over the long haul. Even improbable correlated faults become probable given enough time [10]. Once that threshold is breached, for however brief a period, the system’s fault assumption is violated, and correctness can no longer be guaranteed at any point in time thereafter (Section 2.1).

In this work, we focus on storage applications with a long-term horizon and design a replicated service model that suits them. We observe that the reliability of a system’s components over long spans of time can vary dramatically. First, a complex but formally unverified software artifact might be likely to exhibit vulnerabilities; all that stands between it and a bug is a lapse in the judgment of a human programmer. However, a formally verified software artifact might take much longer to exhibit vulnerabilities: it will not exhibit bugs against which it was verified, but perhaps the assumptions under which its correctness was verified might cease to hold upon a radical technology change (think of the transition from uniprocessors to multiprocessors as such a change). Taking this rationale to its extreme, a trusted third party—a “component” in a distributed service, such as a root DNS server—might take even longer to fail: for example, even if all involved hardware and software components are operating as specified, the trusted component can fail if the organization operating it sells out to criminals. We argue that whereas this differentiation might be esoteric and moot for near-term services, it may be an unavoidable consideration for long-term applications.

This observation leads us to a *tiered fault framework* for such replicated applications (Section 2.2). This framework partitions system components into different classes; for instance, software and hardware used for write operations is in a different class from software and hardware used for read-only operations. The framework treats components of one class across all nodes separately from components from another class, assigning a separate *fault tier* to each class. Like more traditional models, the fault assumption within each tier is threshold-based, but the actual threshold differs from tier to tier. For instance, the fault assumption for the population of write-operation components may be a $1/3$

threshold as with typical BFT systems, whereas the fault threshold for the population of read-operation components may be higher. There is no magic in this formulation: each fault tier is itself subject to a fault threshold. However, this multi-tier approach enables us to structure a system so as to operate longer without violating its overall fault assumptions.

One could informally view this tiered fault framework as a “reallocation of the dependability budget” across the different hardware and software components and across time. This stronger fault framework implies different operational practices for each component class: high-trust components must be formally verified before deployment—which might imply that they be limited in functionality or that they run infrequently and briefly, and are mostly off-line to reduce their attack surface—whereas lower-trust components might be larger, bug-gier, and running continuously.

To make things concrete, we study a particular kind of long-term application: an authentic long-term key-value store. Such a facility can be useful, for instance, as a directory for finding sensitive data given a human-memorable name. One example is a directory for self-certifying names of stored files given a file’s human-friendly name. Such a service can close the loop for previously proposed reliable long-term archival services such as Glacier [25], Oceanstore [31], Pergamum [49], and Preservation DataStores [20], which can withstand Byzantine attacks only as long as a client holds a self-certifying name for a data item. This leaves out of scope the task of *finding* those human-unfriendly names by a user in the future, not to mention that today’s self-certifying names (e.g., the SHA-1 hash of a document) will not be certifying anything in the future if the technology behind them is defeated (this was recently demonstrated as inevitable for SHA-1 hashes [19]).

Bonafide is such a key-value store that provides its correctness guarantees (integrity and liveness) under a tiered fault model (Section 3). Bonafide partitions the operation of a key-value store into three classes of components bound by three tiers of threshold-based fault assumptions. The lowest, most error-prone tier of Bonafide is occupied by the *service process*, a mechanism for responding to the clients’ read-only requests (e.g., looking up existing key-value bindings) and for buffering—but not executing—new key-value additions. The middle tier contains the *update process*, which performs in batch all buffered key-value additions, but runs periodically and only briefly. The highest tier contains a minimal trusted facility for a *moded, attested storage* module (MAS), which keeps the error-prone service process safe and protects the integrity of the update process.

Bonafide provides its guarantees as long as no component of the trusted top tier and fewer than a third of

middle-tier components fail at the same time; any number of bottom-tier components can fail. In addition, like other systems such as Carbonite [15], Bonafide offers durability (that is, does not lose stored key-value bindings) as long as the system creates copies of data faster than they are lost.

Our prototype implementation provides a simple add/get interface and shows reasonable performance (Section 4). We note that most building blocks for Bonafide are borrowed from prior work, most notably from trusted primitives, authenticated data structures, proactive recovery, and BFT replicated state machines. It is the structuring of Bonafide as a service observing a tiered fault framework for long-term operation that we claim as novel.

We discuss the tradeoffs and extensions of Bonafide in Section 5, describe related work in Section 6, and conclude in Section 7.

2 Tiered Fault Tolerance

In this section, we demonstrate how a uniform-fault-threshold system model is not suitable for long-term applications, and we introduce a tiered fault framework examining its feasibility for long-term applications. Although we focus here on Byzantine faults, the approach applies to weaker kinds of faults as well.

2.1 Fault Assumption Violations

We give here an example of how a system built on Castro and Liskov’s popular Practical Byzantine Fault Tolerance (PBFT) [13] protocol for replicating state machines breaks with even a transient violation of the fault threshold. In PBFT, an upper bound f on the number N of replicas allows the use of replica quorums (typically of size $2f + 1$) to protect the safety and liveness of the system, only as long as $N > 3f$. Figure 1 illustrates a population of $N = 4$ replicas, of which r_1 and r_2 are faulty, in violation of PBFT’s fault bound¹ ($f = 1$). Furthermore, non-faulty replicas r_0 and r_3 cannot temporarily communicate with each other, e.g., due to transient interference such as DoS from the faulty replicas. Client a sends req_a to the system. The two faulty replicas convince r_0 to commit and execute req_a first, since the three of them form a quorum of $3 = 2f + 1$. Later client b sends req_b to the system. The two faulty replicas convince r_3 to commit and execute req_b first, since r_3 never saw req_a . Henceforth, all results that the two non-faulty replicas send back to clients will be dependent on divergent views of the system’s global history and state.

Because only $2(= f + 1)$ matching replies are required to convince a client of a result, even if the fault assumption is again met because one faulty replica is repaired, the remaining one faulty replica will always be able to corroborate r_0 ’s view of the world to some

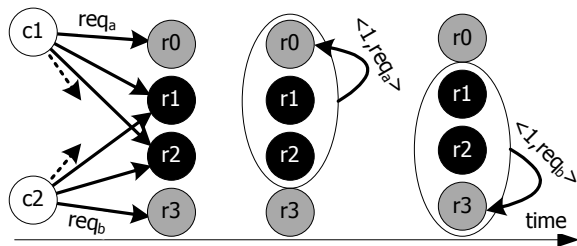


Figure 1: An example that shows the potential effects of a fault threshold violation in PBFT. Black circles are faulty replicas (one of them is the primary), gray circles are correct replicas, and white circles are clients. When two clients c_1 and c_2 submit requests req_a and req_b to the replicas at roughly the same time, but only manage to reach one correct replica each, the two faulty replicas can convince the two correct replicas to assign the same sequence number to different requests.

clients and r_3 's view of the world to other clients, keeping up the charade indefinitely. Crash-fault tolerant replicated state machines based on the Paxos [32] protocol do not deal with Byzantine faults explicitly (i.e., assume a Byzantine-fault threshold of 0) and they can have similar problems if a Byzantine fault crops up rarely and briefly.

Though not general for all possible replicated state machine protocols, this illustration serves to demonstrate the common trend: once the fault assumption is violated (the same as a threshold breach in traditional BFT protocols) the system cannot offer its correctness guarantees again, even if the fault assumption is later restored.

The fault bound in the original PBFT protocol applies over the lifetime of the system, assuming that once a replica becomes faulty it does not recover. PBFT's authors devised PBFT-PR, an enhanced protocol with some hardware support that attempts to repair faulty replicas. As a result, PBFT reduces the length of the *vulnerability window* of the system during which the fault bound might be breached; even though more than f faults may occur during the lifetime of the system, as long as faults are repaired frequently enough so that no more than f faults are ever simultaneously present, the system maintains its guarantees.

PBFT-PR achieves this repair using *proactive recovery* [14]: a hardware watchdog on every replica periodically reboots it with a fresh software installation from a read-only medium (e.g., a CD-ROM), flushing any runtime code damage caused since the last reboot. Upon reboot, the protocol cleans up the service state before it goes back into regular operation. Now the window of vulnerability² is the period of time between two successive, successful proactive recovery phases across the replica population, which is much shorter than the lifetime of the system. However, if the f fault bound is violated within a vulnerability window, the protocol fails once again.

2.2 Tiered Fault Framework

We observe that the traditional fault model mostly presents an either-good-or-bad view of the world. Nodes that are faulty are incorrect and there is nothing in between. In reality, however, different components of nodes in a complex system exhibit different levels of fault tolerance; this fact is also explored in the wormholes model for short-term applications [51, 53] and we compare our approach with the wormholes model in detail in Section 6. In this work we argue that it may be unavoidable for long-term services to use a *tiered fault framework*, which exploits different levels of reliability in different components of the system. In this framework, different classes of components of the service implementation are assumed to keep their numbers of (Byzantine) faults under different thresholds.

We believe that a tiered fault framework is desirable because of the broad differentiation among software and hardware components. One source of differentiation comes from *different assurance practices*. Hardware microprocessor designs undergo extensive formal verification before production and, though extremely complex, tend to exhibit fewer bugs and security vulnerabilities in their implementation than typical software systems. Even in the software world, formally verified components can rigorously prove their correctness guarantees under specific execution models and, as a result, be protected from many runtime bugs and vulnerabilities [40, 45]. This approach can be leveraged with some success and performance cost via the use of strongly typed languages such as Java and C#, which are touted as safer environments for building robust systems: they offer a formal guarantee that, as long as the execution runtime implements the language semantics correctly, no application will be vulnerable to some of the typical system plagues like buffer overflows. Similar guarantees are offered by language-based type-safe operating systems such as Singularity [27].

A second source of differentiation comes from *care in the deployment of a system*: tight physical access controls, proactive hardware and software replacement, responsive system administration, well-designed firewalls and intrusion detection mechanisms contribute to keeping out the threats that can exploit any vulnerabilities present in the physical and logical interfaces of a system component. For example, a software component that is vulnerable to a particular exploit borne over SSH traffic can be shielded from that exploit if the firewalls between the Internet and that component drop all SSH packets before they reach it [54], or if it only communicates with other trusted components over a private network [18, 46].

A third source of differentiation comes from the *rolling procurement* characteristics of the software and

hardware technologies in long-term services. Unlike typical “near-term” systems, it is not the data that “flow” through the hardware and software, but instead the hardware and software that “flow” through the long-term service data³: though the service needs to remain the same, hardware becomes obsolete, operating systems evolve, communication standards grow, and cryptographic best-known methods are broken and replaced by their successors. For example, a trusted logical component assumed to never fail would require the expensive proactive replacement of the cryptographic libraries or the trusted hardware platform used to implement it, as new cryptanalysis techniques become possible, faster hardware is introduced, and new processes for protecting processor packages from physical or electrical tampering become available. In contrast, a less trusted component could afford to trail the state of the art and use replication or other techniques to mask faults, only migrating to new software and hardware less frequently and potentially at a lower cost.

Finally, fault differentiation can come from *limited exposure*. Many high-assurance systems such as certification authorities keep their sensitive components (e.g., their signing keys) mostly or wholly off-line, limiting attack opportunities. Services that have limited or potentially batched updates but can be mostly read-only (or indeed off-line with on-line, untrusted proxies [21,30,33]), can be protected quite effectively in this fashion.

Interestingly, there are non-trivial dependencies among all these sources of differentiation. For example, a proven-correct system that is operated by a trustworthy organization is strictly more reliable than the same system operated by an unreliable organization. As goes the usual secure systems’ truism, a complex system is as secure as its weakest link. This simple observation allows us to argue that long-term fault models can usefully and realistically be constructed in which different system components belong in different *fault tiers*: in each tier, a different fault threshold can be assumed, though the justification for that fault threshold might imply restrictions on the component capabilities for each tier. For example, if one were to argue that a component tier is afforded a low fault threshold thanks to its being formally verified, that component cannot be too complex: formal verification is still an extremely expensive proposition both in terms of human effort and computational resources [56]. Similarly, a tier whose fault threshold is justified by its remaining mostly off-line had better correspond only to functionality that the service can afford to perform periodically in batches.

3 Bonafide

Our target application in this paper is Bonafide, a key-value store designed to provide long-term integrity us-

ing replication in a tiered fault model. We are motivated to build a long-term key-value store not only as a case study for the system-building approaches we described earlier in the paper, but also because it is fundamentally needed by archival storage systems such as Glacier [25], OceanStore [31], Pergamum [49], and Preservation DataStores [20]. Whereas such systems provide durability (protection from data loss) and authenticity, they require data to be *self-certified* for their authenticity properties to hold: a client who needs to fetch a document from the archival system must have an authenticator such as a cryptographic hash of the document’s contents to ensure that what the service returns has not been modified; a client who does not have such an authenticator cannot obtain any authenticity guarantees from the service. We seek to create an archival service for providing indirection for precisely such authenticators: it can be used as a lookup service from a URL or a human-readable name to the random bits making up the authenticator, which can then be used to fetch the actual document from Glacier, Oceanstore, or systems similar to them.

In the simplest case, a system like Bonafide offers a minimal interface: clients invoke Bonafide’s `Add(key, value)` method to store and preserve a particular key-value pair, if no such key is already being preserved, and the `Get(key) → value` method to obtain any stored key-value pair by that key, if one exists. The service is append-only. There is no method to remove or replace an existing key-value pair. We use an append-only interface for simplicity; it is not a requirement.

3.1 Design Rationale

We apply the intuition behind the tiered fault framework by attempting to refactor the functionality of a service such as Bonafide into a more reliable fault tier for state changes and a less reliable fault tier for answering read-only state questions (i.e., `Get` requests). In keeping with the justification for distinct fault tiers, we make the reliable, state-changing functionality mostly off-line, running periodically to execute state changes in batch (for `Add` requests buffered but not executed during the mostly unreliable operation of the system).

One challenge with such a high-level design, especially when using commodity hardware, is the isolation between the reliable and the unreliable class of components. The Castro and Liskov approach punctuates a system’s timeline with periodic software refreshes, which can help bring a system whose faults are climbing towards the fault threshold back from the precipice. Unfortunately, that isolation goes away once the system has crossed the precipice; even if the number of faults is somehow reduced below the fault threshold again, data

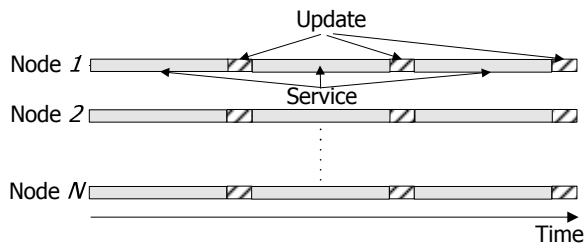


Figure 2: Operation of Bonafide. Each Bonafide replica alternates between a service phase (S phase) and an update phase (U phase).

changes before and after the fault violation cannot be isolated, causing the loss of safety and liveness guarantees.

To address this challenge, we require our third, most reliable class of component in its own top fault tier: a trusted mechanism for protecting state during execution of the unreliable class of components. This mechanism is not only extremely critical, but cannot even be mostly off-line; as a result, to ensure its fault threshold is plausible, we must make it extremely simple. For Bonafide’s top-tier component class we use a *moded, attested storage* (MAS) facility, akin to the sealed storage mechanism provided by modern trusted platform hardware [4]. This facility allows us to store reliably very small amounts of memory, only allowing the reliable stage-changing mechanism to update this storage.

A second challenge is that our middle-tier, reliable state-changing mechanism must somehow be able to mask faults experienced by its components. We use a Byzantine-fault tolerant replicated state machine approach to implement this middle-tier mechanism. However, since this middle tier is mostly off, we require that all of the system’s nodes execute this tier in a synchronized fashion, which implies loose clock synchronization and a fairly long period for the execution of this tier. We describe how to relax the requirement for clock synchronization and synchronized execution of the state machine replication in Section 5.2.

The final challenge is figuring out how to use our very limited attested storage to protect a potentially large service state. The approach we adopt is the use of an authenticated data structure, which allows the integrity of arbitrarily large, structured data to be protected by a small cryptographic digest.

3.2 Overview

Bonafide is a replicated service running on replicas $R = \{1, \dots, N\}$ ($N = 3f + 1$). The replicas operate in alternating synchronous phases of two types: a *service phase* (S phase) and a subsequent *state-update phase* (U phase) (Figure 2). During the i -th S phase, Get requests can query the service state committed (i.e., fetch bindings

Fault bound	Component	When	How used
0	Watchdog	Periodic	Invoked
	MAS	S phase	Read
		U phase	Written/Read
1/3 Byzantine ⁴	Update	U phase	Replicated store Serve ADDs
Unbounded	Service	S phase	Serve GETs Buffer ADDs Audit and repair

Table 1: The components in Bonafide and their associated fault tiers.

that were added) up to the $(i - 1)$ -st U phase. Add requests are buffered and executed after the end of the i -th S phase, during the i -th U phase. In other words, service state changes occur in batch *only* during the U phase.

Bonafide consists of three component classes (trusted storage, state update, and service), each of which belongs to a fault tier with a different fault threshold. Table 1 summarizes the fault tiers in Bonafide. The state update component of a replica contains the state update process, OS, and hardware excluding the trusted top tier, and the service component of a replica contains the service process, OS, and hardware excluding the trusted top tier.

In addition, Bonafide has the following standard, partial synchrony assumption for liveness. In the network, packet drops, reorderings, and duplications can occur but retransmissions of a message eventually deliver it. However, though finite upper bounds exist for message delivery and operation execution times, those bounds are not known to protocol entities. This is a standard network assumption for Byzantine-fault tolerant systems and is not unique to Bonafide.

Under this tiered fault assumption, Bonafide guarantees service safety, that is, *integrity of returned data*. However, to guarantee *durability* as well (i.e., that no data are lost) the system should create copies of data faster than they are lost, as in systems such as Carbonite [15]. Also, to ensure *liveness* (i.e., non-starvation) S phases with at most $2/3$ faulty replicas must occur once in a while (more precisely, within a finite number of phases at all points in time). This is to ensure that an Add request must be resubmitted by a client a finite number of times before it is eventually served by a U phase.

A Bonafide node contains a MAS as well as a buffer to hold Add requests temporarily and a main data structure that maintains committed bindings (Figure 3). In Bonafide, the service state—the key-value pairs—is maintained as a variation of a hash tree [37], which computes a cryptographic digest of the whole state from the leaves up, storing it at the tree root. The results of individual state queries (i.e., key lookups in the tree) can be validated against that root digest; as long as the digest is kept safe from tampering, individual lookups can be performed by an untrusted service component with-

out risking an integrity violation. This state is replicated at each replica in the system in untrusted storage (bottom tier) but its root digest (of size on the order of 1 Kbit in today's hardware) is stored in each node's MAS. Each replica's MAS module lies in its most trusted fault tier: we assume that while in service no MAS module returns contents other than those that were stored at it. We use a MAS for the root digest of the service state, since it cryptographically protects the integrity of any answers about that state provided by even an untrusted component.

The service state is updated during the U phase, which is invoked by a trusted watchdog in the most trusted fault tier. In the U phase, all buffered writes are agreed upon by non-faulty replicas using a state machine replication protocol and then reflected in the service state, replacing the integrity digest in each replica's MAS. The U phase is in the next most trusted fault tier in Bonafide: we assume that no more than a third of the replicas' update software can fail simultaneously, to ensure that the state machine replication protocol safety and liveness guarantees can be upheld within a single U phase.

The service state is served to clients during the S phase. Responses to Get/Add requests are accepted by clients when $f + 1$ replicas return to the client the same result, and each result is consistent with the corresponding replica's service state digest in its MAS module. The $f + 1$ number comes from the fault bound of the update tier, which assumes no more than f update processes can be faulty in any single U phase; as a result, no more than f update processes can put an incorrectly updated digest into their own MAS. If the same response to a client request is provided by at least $f + 1$ untrusted service processes, but backed by the $f + 1$ trusted state digests in the MAS, the client is guaranteed to be getting what at least one correct replica provides. At worst, the client will receive no valid responses or obviously invalid responses from the replicas and try again. Also, the service state is audited (for latent storage faults or other bit loss) and repaired during the S phase.

At the protocol level, Bonafide provides the following safety property. If an Add or Get operation collects $f + 1$ matching replies from distinct replicas, the reply is correct. In other words, there is a serial schedule of committed bindings, and once a binding is committed, it is seen by clients. This is similar to the integrity guarantees of other long-term storage systems, but unlike them, Bonafide can take any key to "name" the sought data value, not only self-certified names. In this paper, we discuss only Adds, but the safety property of Bonafide holds even when there are Removes or Replaces in the system API.

In addition to safety, Bonafide provides the following liveness property. If an Add operation collects $2f + 1$ tentative acknowledgments from distinct replicas, the bind-

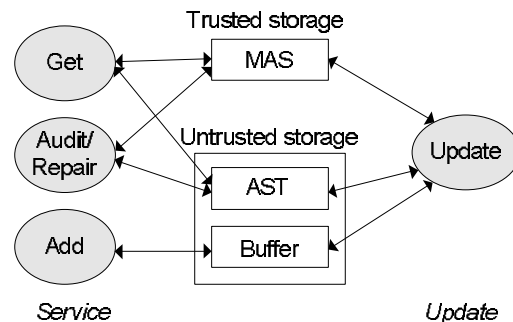


Figure 3: A Bonafide node contains the following state shown in the middle of the figure: a MAS, a buffer to hold Add requests temporarily, and an AST that maintains committed bindings. The MAS stores the AST root digest, a sequence number, and a checkpoint certificate. The left side shows the get, add, audit/repair processes running during the S phase, and the right side shows the update process running during the U phase. The arrows show what state the processes access.

ing is guaranteed to be committed during the U phase if there are at most f faults in the S phase during which the operation is invoked. If there are more than f faults in the S phase, the Add operation does not guarantee the binding to be committed during the U phase since faulty replicas can send fake tentative acknowledgments.

A fundamental limitation of Bonafide is that it is not resistant to common denial-of-service attacks, such as *name squatting*. As in all existing archival systems we are aware of, faulty clients can insert arbitrary bindings into Bonafide, preventing legitimate clients from using those bindings.

Next we detail the service state and component functionalities of Bonafide.

3.3 Bonafide State

In Bonafide, the service state (the collection of key-value bindings) is maintained as an authenticated search tree (AST). An AST [11] is an incremental mechanism for maintaining cryptographic digests over sorted data sets (such as key-value pairs sorted by key), extending the concept of a traditional Merkle tree [37] for search. Every node contains a key-value pair and an authentication label. The label for an AST node is computed by hashing together its content and the labels of all child nodes. The label of the tree root is a cryptographic digest for the entire contents of the tree: it is *collision-resistant*, which means it is intractable to find two different data sets yielding the same AST digest and, as a result, it can serve as a *commitment* on the contents of the AST [39].

As with Merkle trees, a succinct *witness* (sometimes also called a *proof*) can be generated showing that a particular key-value pair appears within an AST with a root label. Unlike Merkle trees, an AST can also provide a succinct witness that a key *does not appear* within it.

Witnesses have logarithmic length in the number of the key-value pairs contained within a tree.

Each Bonafide replica maintains an AST in typical (untrusted) storage containing its collection of key-value pairs sorted by key, a buffer of received but yet uncommitted client requests for adding new key-value pairs also in untrusted storage, and a MAS having two slots within its trusted hardware device (see Section 3.4). The MAS slot with identifier q_s stores values of the form $\langle s, r \rangle$, where s is the latest AST digest and r is an integer sequence number associated with a particular U phase. The slot with identifier q_c stores a checkpoint certificate described in Section 3.6. Finally, replicas know each other's public keys and hardware device public keys.

3.4 Top Tier: Trusted

Cryptography: We assume standard cryptographic primitives for digital signing and hashing. These primitives belong in the top tier of the fault model (as they do in virtually all systems research and practice). In keeping with our “rolling procurement” argument for this trust over long periods (Section 2.2), we assume that cryptography is replaced with up-to-date technologies, algorithms, and key sizes well before its compromise is even feasible let alone practical (see Section 5.2 for a sketch of how this is accomplished). For a managed system, this is a reasonable and plausible approach.

For conciseness, we denote by $\langle M \rangle_i$ a message M that is digitally signed by principal i . Replica i 's trusted hardware device (i.e., its MAS principal) is principal i' . If M is not signed, there is no subscript in the notation.

Trusted Hardware: Bonafide relies on the existence of a trusted hardware device on every replica in the system [8]. Today, such a device could be implemented in a programmable, tamper-resistant secure coprocessor such as IBM's commercially available PCIXCC [8] board, but cheaper trusted alternatives are implementable with the trusted computing platforms coming from Intel's (e.g., Intel TXT) and AMD's (Presidio) pipelines.

To help conduct periodic recovery operations, this device contains a time source (this can be a regular, monotonic, crystal-based clock source with an upper bound on drift, or an external trusted time source received by the device such as GPS). A hardware watchdog, also contained within, uses this time source to trigger proactive recovery periodically, by causing the host to reboot from read-only media. This hardware watchdog sets a *mode* bit of the MAS associated with it. This bit is used to indicate that the system is in its U phase, and cannot be set in any fashion other than by triggering the watchdog. The mode bit can, however, be reset by the operating system. Typically this is done during the U phase, while the software is still under the middle trust tier. During the S phase, the no longer trusted operating system can of course reset

this bit, but bit resets are idempotent, so this misbehavior is ineffective. Such mode bits are sometimes called *sticky registers*.

Finally, the device contains a MAS with a simple interface. A MAS contains a mode bit, and a set of storage slots, each of which is identified by an identifier q . The write interface to a MAS is $\text{Store}(q, v)$ where v is a value; this stores v to the slot with identifier q . This interface allows requests only when the mode bit indicates a U phase is ongoing. The read interface of MAS allows access all the time. It allows the attested, fresh retrieval of any slot; a $\text{Lookup}(q, z)$, where q is a slot identifier and z is a nonce used for freshness (typically provided by clients), returns $\langle \text{LOOKUP}, q, v, z, t, m \rangle_{i'}$, where v is the value currently occupying the slot with identifier q of the MAS, t is the internal time in the device, m is the current mode bit, and i' is the hardware device principal. If the slot is empty, then $v = \text{EMPTY}$ in the returned attestation. In our own recent work, we have introduced an Attested Append-Only Memory (A2M) [16] and we compare MAS with A2M in detail in Section 6.

Membership Management: Bonafide is intended for a well provisioned, low-churn node infrastructure. Since membership churn is low, the membership of nodes can be managed manually. Membership management is conceptually also trusted, in the top tier of the fault model. We discuss how to extend Bonafide to automate membership management in the middle tier by refactoring it with MAS in Section 5.2.

3.5 Bottom Tier: Service Process

In the S phase, each Bonafide replica runs an add/get process to serve client requests, and a continuous audit and repair process in the background for durability. Pseudocode for the service process is given in Figure 4.

Get: When a client c invokes $\text{Get}(k)$ to retrieve a value of key k , its Bonafide stub (called a *proxy* below) multicasts $\langle \text{GET}, k, z, c \rangle_c$ messages to R where z is a nonce used for freshness and waits for $f + 1$ $\langle \text{REPLY}, i, v, p_i, \langle \text{LOOKUP}, q_s, \langle s_i, r_i \rangle, z, t, m \rangle_{i'} \rangle$ valid matching messages confirming that (k, v) is within the AST, or that (k, v) does not exist in the AST. Note that the attestation includes the nonce the client sent to ensure it does not accept a stale response.

A replica handles a GET by looking it up by key in its local AST and producing an existence/non-existence witness, accompanied by its latest MAS attestation.

Add Buffering: When a client c invokes $\text{Add}(k, v)$ to insert a binding between key k and value v , the Bonafide client proxy code multicasts $\langle \text{ADD}, k, v, z, c \rangle_c$ to R . The client waits for $2f + 1$ *tentative* acknowledgments, each of which is a $\langle \text{TENTREPLY}, k, v, z, c \rangle_i$ message where i is a replica identifier, from distinct replicas. If the client

```

CLIENT.GET(key)
// quo_RPC sends msg to R, collects matching responses on non-*
// fields from a quorum of given size, retransmits on timeout
⟨REPLY, *, value, witness, *⟩ ← quo_RPC(⟨GET, key⟩, f + 1)
return value
SERVER.GET(client, key)i // this is server i
⟨value, witness⟩ ← lookup_AST(key)
att ← lookup_MAS(qs) // attestation
send client a ⟨REPLY, i, value, witness, att⟩

CLIENT.ADD(key, value)
⟨TENTREPLY, *, key, value⟩ ←
  quo_RPC(⟨ADD, key, value⟩, 2f + 1)
// at this point, the client holds a tentative reply
collect REPLY messages // in the next S phase
if (f + 1 valid, matching replies are collected)
  return accepted(key, value)
SERVER.ADD(client, key, value)i
if (⟨key, value′⟩ in AST), treat as a GET and return
add ⟨client, key, value⟩ to Adds
send client a ⟨TENTREPLY, i, key, value⟩

SERVER.AUDIT(ASTNode, hASTNode)i
status ← check ASTNode, hASTNode
if (status invalid) repair ASTNode // fetch from other
for each child C of ASTNode
  AUDIT(C, hC) // hC is contained in the label of ASTNode

SERVER.START_SERVICE(Committed_Adds)i
// reply for ADDs committed in the previous U phase
for each ⟨key, value, client⟩ in Committed_Adds
  send client a ⟨REPLY, i, value, witness, att⟩

```

Figure 4: Simplified service process pseudocode.

does not receive the responses within a timeout, it presumes that the request has been dropped by the network, so it retransmits the request to the replicas that did not respond. Note that receiving $2f + 1$ tentative acknowledgments is a hint that means the binding is likely to be committed. The client does not perform any operation that depends on the fact that the binding is committed and cannot be undone. Our liveness guarantee ensures that the client will receive a final commitment (see below) for each Add after receiving a finite number of uncommitted $2f + 1$ such hints.

The client also waits asynchronously for *commit* replies in MAS-attested messages of the form ⟨REPLY, *i*, *v*, *p_i*, ⟨LOOKUP, *q_s*, ⟨*s_i*, *r_i*⟩, *z*, *t*, *m*⟩_{*i*}′⟩ (the attestation is the result of a MAS Lookup). These messages are sent by replicas in the beginning of the next S phase. A reply is valid if witness *p_i* verifies the existence of the key-value pair (*k*, *v*) within an AST with digest *s_i*, and the attestation is correctly signed by the sender’s MAS. As soon as the client proxy obtains *f* + 1 valid matching replies from distinct replicas, all confirming the addition of the same key-value pair, it accepts the request as *complete* and notifies the application.

During the S phase, a replica only buffers ADDs and sends a TENTREPLY message back for each ADD. It handles the ADDs during the U phase. The replica also re-

```

SERVER.UPDATESTART()i
  PBFT.Invoke(⟨BATCH, i, stable_ckpt_cert, Adds⟩)i
  SERVER.FINALIZE()i

SERVER.EXECUTE(batch)i // PBFT Execute callback
append batch in batch_log
on receiving the 2f + 1-st batch:
  choose the latest stable_ckpt from batch_log
  AllAdds ← the union of the Adds set from each batch
  for each ⟨key, value, client⟩ in AllAdds
    repair the AST path to this new key if needed
    insert key, value into AST
    insert ⟨key, value, client⟩ into Committed_Adds
  store_MAS(qs, ASTRootDigest)
  multicast a UCHECKPOINT and flush batch_log

SERVER.FINALIZE()i
on receiving 2f + 1 matching UCHECKPOINTS:
  store_MAS(qc, stable_ckpt_cert)
  reset the watchdog timer and the mode bit
  begin a new S phase

```

Figure 5: Update process pseudocode.

turns the existing mapping for ADDs for already assigned names. During the next S phase, the replica responds to newly inserted ADDs with a REPLY message. The average (committed) response time for ADDs of new bindings is on the order of the S phase length.

Audit and Repair: The audit and repair process ensures that all reachable AST nodes from the AST root are correctly stored. This process is recursive, starting with SERVER.AUDIT(*ASTRoot*, *h_{ASTRoot}*) where *h_{ASTRoot}* is the digest of the AST root and traversing the tree in order, during which a tree node is fetched from storage if still available and verified by computing its hash value and comparing it with the hash contained in the label of its parent node.

For every missing AST node with digest *h*, replica *i* multicasts a ⟨REQASTNODE, *i*, *h*⟩_{*i*} request to *R*, waiting for at least one ⟨RESPASTNODE, *h*, *ASTNode*⟩ response. The response need not be signed, since the replica can verify its integrity thanks to the recursive hashes of the AST. As long as the root digest remains in the trusted MAS, the rest of the AST nodes are self-certifying.

3.6 Middle Tier: Update Process

When the trusted watchdog timer expires, the system begins a reboot securely from a read-only medium of its proactive recovery software. The main responsibility of the U phase is to commit a new set of additions into the main service state. At the end of the U phase, the system ensures that at least $2f + 1$ replicas store the latest service state digest in MAS (see Figure 5 for the pseudocode).

We use the PBFT protocol [14] to replicate the state machines of individual U phases, though any BFT state machine replication protocol would work. PBFT offers a synchronous Invoke(*request*) method, that returns a *response*. A PBFT client (which is a replica’s U phase

in our use of the protocol) uses this method to submit application requests—buffered `Add` requests—to replicas and eventually receives replies containing the result. PBFT also offers replicas an `Execute(request)` callback, which invokes the application code that processes ordered client requests to be executed—the actual insertion of `Added` key bindings into the service state.

All messages exchanged between replicas contain a fresh attestation fetched from the MAS after the current U phase began: the mode bit shown must be on, and the timestamp must be recent. Messages unaccompanied by this attestation are invalid and dropped. This is to ensure that update operations, including invocations of PBFT, are performed by nodes that have rebooted into their U phase. Any faults caused by such nodes are due to update process faults, which our middle fault tier bounds by f .

Update Start: Each replica packages up its pending ADDS (denoted by A) and the latest stable checkpoint (i.e., $2f + 1$ matching MAS attestations of the previous round) (denoted by C_s) obtained from the MAS slot q_c into a $\langle \text{BATCH}, i, C_s, A \rangle_i$ message, which it submits to PBFT's `Invoke`.

Application State Machine: The update state machine, executed on `BATCH` requests as ordered by PBFT, stores `BATCH` requests in order, until it receives the $2f + 1$ -st of them (from distinct replicas). Subsequent batches are ignored to ensure liveness (there is no way for replicas to know how many batches they can expect beyond the $2f + 1$ -st). Note that if there are at most f faults during the S phase, there is at least one correct replica submitting every request whose client received $2f + 1$ tentative acknowledgments during the S phase, precluding starvation. As long as there are such S phases with no more than f faults from time to time, the system makes progress.

In receiving the $(2f + 1)$ -st batch, the application state machine picks the latest stable checkpoint, and the union of all ADDS across all $2f + 1$ batches. It orders the ADDS according to a consistent order (e.g., by $h(k \| v \| c)$), verifies the client's signature, and inserts all valid pairs into the AST in that order, ignoring keys that already exist. The replica computes the new AST digest s_i^* for sequence number $r_i^* (= r + 1)$, stores it into the q_s slot of its MAS, and multicasts to R a $\langle \text{UCHECKPOINT}, \langle \text{LOOKUP}, q_s, \langle s_i^*, r_i^* \rangle, t, m \rangle_{i'} \rangle$ message.

When a replica receives $2f + 1$ matching `UCHECKPOINT` messages, it stores the set at the MAS slot q_c as a new stable checkpoint certificate. If a replica's old service state is not the latest one, it will have to perform state transfer, as described below.

When the replica obtains a new stable checkpoint certificate, it resets its watchdog timer to \mathcal{D} , which is the remaining time until the next U phase, and exits into its S phase by opening up communication with nodes other

than replicas and resetting the mode variable. In the beginning of the new S phase, the replica sends `REPLY` messages for all newly inserted ADDS as described earlier.

State Transfer: Up-to-date replicas missing actual service state (e.g., because some of the AST nodes were corrupted) can apply the same repair process used during the S phase to obtain the AST nodes required for their batched ADDS.

Before the phase can end, the MAS of a replica must contain the latest stable checkpoint. A slow replica may be behind to obtain its checkpoint by executing the agreed-upon write operations. However, the stable checkpoint broadcast by those replicas that were up to date allows a slow replica to append that state digest into its MAS, thereby catching up with others and entering to the next S phase.

Single-agreement Optimization: The design described requires at least $2f + 1$ PBFT invocations, one per `BATCH`, during every U phase. In the worst case, each invocation requires 3 network roundtrip times, potentially increasing the latency of the U phase tasks, which increases the minimum duration of the U phase, which in turn reduces the availability of the system. Instead, the update process can do preprocessing to create a `PROPOSE` message containing at least $2f + 1$ `BATCH` messages and only submit that proposal to PBFT. This optimization duplicates the functionality of the PBFT primary by introducing a leader to collect the `BATCH` messages. At the cost of greater complexity, this optimization can make use of all available `BATCH` messages, not just the first $2f + 1$ messages generated by replicas, and also reduce the worst-case number of roundtrip times required. Our implementation uses this optimization.

Each replica packages up its pending ADDS A and the latest stable checkpoint C_s obtained from the MAS slot q_c into a $\langle \text{BATCH}, i, C_s, A \rangle_i$ message, filtering out those ADDS for already assigned keys, and broadcasts the `BATCH` to R . Once a *leader* replica (defined below) collects at least $2f + 1$ such messages including its own, it packages them into a `PROPOSE` message, which it submits to PBFT's `Invoke` for Byzantine agreement. During the `Execute` callback of PBFT, a replica ensures the `PROPOSED` set contains at least $2f + 1$ batches from distinct replicas. If so, it runs the function to update the service state, and the rest of the U phase is the same.

During each U phase, the leader described above is the replica ($l \equiv r \bmod N$), where r is the current U phase round number. A leader may misbehave, either by delaying the transmission of a `PROPOSE` message, or by transmitting an incorrect such message. The latter case can be detected during the `Execute` PBFT callback, as described above. A non-faulty replica can detect the former case by setting a timer as soon as it multicasts its `BATCH` message, which it uneventfully stops when it encounters

its own BATCH as one of the batches included in a proposal during the `Execute` callback; if the timer expires, then the replica initiates a leader change. To avoid unnecessary leader changes due to transient slowness, the replica does exponential backoff for consecutive leader change initiations.

Leader change is similar to proposal formation: to initiate change, a replica multicasts a `LEADERCHANGE` message, which the *next* leader ($l + 1 \bmod N$) listens for. When that leader has collected $2f + 1$ such requests, it packages them into a single `LEADERCHANGEREQUEST`, which it submits to PBFT; execution of this request increments l , completing the leader change. Note that the leader role is similar but unrelated to the PBFT primary role; PBFT's internal operation, including primary assignment and view changes, is opaque to the U phase functionality.

3.7 Correctness

Under the tiered fault assumption, Bonafide provides the integrity property. Briefly, it is sufficient to show that any binding committed is guaranteed to be safe in the future (if returned, it is the correct binding) and live (if there are at most f faulty replicas, as long as $2f + 1$ replicas have acknowledged receiving the `ADD` request, the binding will be added). We show this by connecting what the client knows ($2f + 1$ tentative acknowledgments) to a starting condition for the U phase, and from there to the steps of the state machine replication. We defer the detailed argument to Appendix A.

4 Experimental Evaluation

In this section, we present the implementation of Bonafide and evaluate its performance.

4.1 Implementation

To validate our design, we developed a prototype Bonafide implementation. We implemented the add/get, background audit and repair, and the optimized version of the update process of Bonafide (excluding leader election) in C/C++ on Fedora Core 6. The client and server communicate with SFS's asynchronous implementation of SUN RPC [47] in the `sfs-lite` library [3]. Client-server communications are authenticated by signatures; we use NTT's `ESIGN` with 2048-bit keys.

The client uses a proxy, its Bonafide local stub code, to perform `Add/Get` operations. The server maintains a MAS, an AST, and a log for buffering `ADDs`. MAS is implemented as a library and it uses NTT's `ESIGN` with 2048-bit keys for signatures as well. We use SHA-1 as a secure hash function.

For Byzantine agreement during the U phase, we use the PBFT library [14] ported to Fedora Core 6. PBFT uses MACs for message authentication. During update,

Operation	Time (ms)	Data loss (%)	Time (s)
	Mean (std)		Mean (std)
Get	3.1 (0.24)	0	554.5 (54.6)
Add	1.0 (0.21)	1	612.9 (30.3)
		10	1147.6 (33.3)
		100	3521.5 (201.6)

(a)

(b)

Table 2: (a) Get and Add time. (b) Audit and repair time.

every node runs an update server and a PBFT replicated state machine. The leader's update server creates a `PROPOSE` message and invokes PBFT agreement on the proposal to get consensus across the population on a hash of the proposal. When consensus is achieved, every replica fetches the proposal from the leader, and validates against the agreed upon hash.

We store an AST and a log using Berkeley DB 4.5.20 [1].⁵ We use a binary AST to minimize the size of membership witnesses [35, 58]. An AST is stored as a Berkeley database with a `BTREE` format. Each AST node is stored as a Berkeley DB record, which contains a key, a value, a hash of its left child, and a hash of its right child. The primary key of this DB is the key, and the secondary key is the hash of the entire node content. To search for a value given a key in the AST and to insert a (key, value) binding to the AST while generating a membership witness, we traverse the AST using secondary keys.

4.2 Performance

We evaluate how the fault-tolerance improvement of Bonafide affects performance and availability. We ran our experiments with four Bonafide replica nodes and one client node. The nodes are outdated PCs with 1.8GHz–3.2GHz Pentium 4 processors, 1GB RAM and 3Com 3C905C Ethernet cards. They are connected over a dual speed 10/100Mbps 3Com switch. On a 1.8GHz machine, `ESIGN` signature creation and verification of 20 bytes take on average $256\mu s$ and $194\mu s$, respectively. We did not opt for a more up-to-date infrastructure since, by its nature, our application need not be deployed on the bleeding edge of hardware. As we see next, even with this obsolete collection of servers and switch, performance is adequate.

Our experiments initially populate server ASTs with one million bindings of a 128-byte key to a 20-byte SHA-1 hash as the value.

ADD/GET Time: We use a simple micro-benchmark client that sends 1000 `ADD` or `GET` requests. For `ADDs`, servers store bindings to their logs and return tentative acknowledgments. For `GETs`, servers search their ASTs and return values, AST witnesses, and MAS attestations.

We measured Bonafide's `GET` and `ADD` response times, by averaging over 1000 requests of each types

Action	Time (s)
	Mean (std)
Reboot	86.6 (2.1)
Proposal creation	8.0 (4.0)
Agreement	5.2 (1.0)
AST update/Checkpoint	271.1 (24.8)
Total	370.9 (24.0)

Table 3: U phase duration with 1000 new committed bindings.

with randomly selected keys. In average, GET takes 3.1ms, and ADD takes 1.0ms (Table 2(a)). GET takes more time than ADD does since it involves accesses along an AST path, which incurs multiple disk block accesses. There is a start-up effect in processing GET requests, roughly 100 requests' long, while Bonafide caches top AST levels.

Audit and Repair Time: We measured the average time of a basic audit that does not perform any repair over five runs. The disk drive we used was an IBM 40GB IDE disk drive with rotation speed 7200 rpm, average seek time 8.5ms, and buffer size 2MB. The mean audit time of the entire AST is 554.5 seconds and the standard deviation is 9.9% of the mean.

To measure audit and repair time, we simulate random data loss. We delete a fraction of AST nodes randomly at a Bonafide replica and run an audit process. When the audit process finds a lost AST node, the process repairs it synchronously by fetching the AST node from a randomly-chosen remote replica. Table 2(b) shows the mean audit and repair time when a fraction of AST nodes are lost. The more data loss, the longer the repair time due to more access to remote nodes.

Note that our current prototype implementation is not optimized. Several optimizations can improve our prototype performance. For example, more intelligent layout of stored key-value bindings may reduce the random disk access [58], thus improving audit time. Also, an audit and repair process can collect missing AST nodes by fetching them in parallel while performing auditing.

U Phase Duration: We measured the duration of the U phase when 1000 new bindings were committed. Table 3 shows the mean and standard deviation of the U phase duration of the leader averaged over five runs; the leader has the highest computation and network bandwidth overhead. Proposal creation indicates time to collect a BATCH certificate and to create a PROPOSE message. Agreement indicates time to run a PBFT agreement, and AST update/Checkpoint indicates time to update the AST with new bindings and to execute remaining COMMIT protocol. Several optimizations can improve the U phase duration. We can reduce reboot time, for example, by using fast boot from the LinuxBIOS project [2]. The project claims three seconds boot time from power-on to Linux console. Intelligent caching of AST nodes may re-

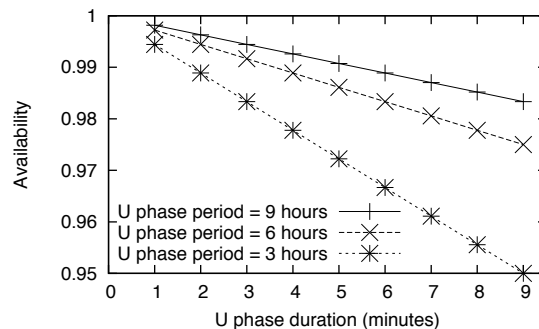


Figure 6: Bonafide availability varying the U phase duration and period. Note that the y -axis starts at 0.95.

duce the AST update time.

Availability: Finally, we analytically show that Bonafide availability (the ratio of service time to service time plus update time) is high enough for varying U phase duration and period in Figure 6. When the update period is 9 hours, availability is 0.998 and 0.983 for one-minute and nine-minute U phase durations, respectively. Availability decreases linearly as update duration increases. In addition, as we perform update more frequently (i.e., update period decreases), availability decreases more rapidly. For example, when update duration is nine minutes, availability drops from 0.983 to 0.950 as update period changes from 9 hours to 3 hours. However, when we perform update frequently, its duration may decrease since fewer additions are collected, mitigating the effects of unavailability. With one-minute update duration, availability becomes 0.994 despite three-hour periods.

5 Discussion

In this section, we discuss the tradeoffs between safety and availability and extensions of Bonafide.

5.1 Safety, Availability, and Durability

Our approach implies an operation model that trades off availability for safety, first by containing state changes during small portions of the system's timeline, and by closing off access to the system by its clients while those state changes are incorporated. Though different applications might fare differently with such a trade-off, we believe that applications like Bonafide, including also notarizing documents [23] and auditing for accountability [24, 58] are appropriate practical candidates.

Bonafide can have tradeoffs between durability and availability. This can be tuned with the frequency of update. Frequent update improves durability since it reduces the probability of N replica faults in an S phase, but it reduces availability since the Bonafide service is not available to clients during update.

Availability can be improved in two ways: (1) somehow removing safely the exclusion of service requests

during U phases and (2) increasing the frequency of U phases without halting the service process.

First, it is possible to run the U phase at the same time as a S phase, if the processes executing each can be adequately isolated. For instance, a combination of virtualization and trusted execution (e.g., Intel's LaGrande technology or AMD's Presidio extensions) can ensure that a new operating system image can be late-launched (i.e., "booted") in isolation of any currently running S phase software in a separate execution domain. While the U phase is running, the S phase executing in a separate domain can still handle requests for the previous snapshot of the service. The same effect could be obtained in perhaps less complexity by separating the U and S phases into different physical machines.

Second, it is possible to increase the update frequency without halting the service process by making update unsynchronized, that is, without requiring that all replicas enter the U phase at the same time. Without the need for clock synchronization among all replicas, the duration of U phases can be much shorter (since we no longer need to accommodate global bounds on clock drift across all nodes) and, as a result, U phases can be much more frequent. We give a sketch of an alternative design for unsynchronized U phases in the next section.

5.2 Extensions

fT -bound: The fault threshold for a single U phase in Bonafide can be extended to a multi-phase fT -bound model, in which the cumulative number of faults in T consecutive U phases is bounded by fT for some fraction f , but there can be phases in which more than fraction f replicas are faulty. Such a failure model may require multi-phase recovery and an extension of MAS to hold, instead of individual registers, an *append-only queue* with at least T positions, akin to an A2M [16].

Early Commitment: Bonafide does not guarantee that a mapping for which a client collects $2f + 1$ tentative acknowledgments in any S phase is committed during the following U phase when there are more than f faults during the S phase. By extending MAS, we can provide early commitment that guarantees a mapping is committed during the following U phase. The attested storage needs to bound the number of the entries appended during a single S phase. Once this storage reaches its bound, it does not accept more appends until it is flushed out during the next U phase.

In early commitment, during the S phase, a replica appends ADDs to the *bounded* attested storage and sends a TENTREPLY message with a MAS Lookup attestation for each ADD. Essentially, the MAS is used as a trusted un-erasable ADD buffer during an otherwise untrusted S phase. As a result, unlike the protocol described earlier, when the client collects $2f + 1$ tentative Add ac-

knowledgements containing buffering MAS attestations, it can complete the request immediately, since that ADD is guaranteed to be reflected in the next AST addition.

Advanced Search: To focus on a tiered fault framework, we present a long-term key-value service with a minimal search interface. Extending the main data structure for advanced search is possible. For example, recent research shows a way for running generalized SQL queries on authenticated databases [21].

Caching for S phases: Bonafide can employ caching replicas that serve Get requests to increase throughput and availability. These caching nodes can serve requests continuously, i.e., they are available during both S and U phases. They use digital signatures that are created by Bonafide replicas to vouch for bindings whose freshness is approximately guaranteed with timestamps. The caching nodes can grow and shrink dynamically depending on workload without manual intervention.

Unsynchronized U Phases: Our current design requires a synchronized execution of all U phases in the entire population, which requires bounds on the drift of all nodes' clocks, which in turn requires a long U phase (to accommodate realistic clock drift bounds). As a result, U phases are infrequent to be mostly off-line.

We are exploring an alternative design that does not require a synchronized execution of all U phases. At a high level, when a replica x in its U phase wishes to send a message m to another replica y that is not guaranteed to be in its U phase, replica x asks y 's S phase to store that message in an untrusted "mail box" for y 's subsequent U phase. When at a later time replica y enters its U phase, it checks its "mail box" for messages from other replicas' U phases, which it uses to make progress.

There are several challenges with this approach. First, the mail box is untrusted (it lives in the address space of the S phase and is subject to the bottom tier of the fault model) which means that messages stored in it may be lost or corrupted. Second, whereas our current, synchronized design executes an entire agreement protocol exchange within a single U phase, this unsynchronized design would have to take multiple rounds of U phases at all involved replicas to complete each agreement protocol (one U phase at each replica to process all messages in its mail box and to transmit the next set of messages). On the other hand, given that there would be no need for clock synchronization, an unsynchronized design can have more frequent but shorter U phases at each replica (say one-second-long U phases every few minutes or so). We are currently adapting a protocol akin to Byzantine Disk Paxos [7], a shared-memory version of Byzantine Paxos that models well communication via unreliable mail boxes.

Upgrades: Bonafide does not require that the cryptographic tools it uses (hash functions, digital signatures)

remain inviolate forever; as long as it is migrated to a new algorithm for hashing or signing before the old algorithm has been completely compromised, it can retain its guarantees. Upgrades require an agreement (via the U phase), using a special `Upgrade` request, handled similarly to `Add` requests (buffered, then committed, then executed). Upgrades can include hardware upgrades (e.g., the migration of one MAS device on a particular replica to a newer device, updating the replica membership and public keys to all those who receive the upgrade), software upgrades (e.g., the installation at a replica of a software module for a new cryptographic function), regular membership updates (switching public keys or locations for a replica), or algorithms in use. The latter case requires that all replicas have the new software for a new algorithm; the system cannot migrate from RSA signatures to a (fictional) new RSA++ algorithm until at least a strong quorum of replicas speak RSA++. As a result, an `Upgrade` request to algorithm RSA++ executes only if all replicas in the membership list already have software for the algorithm; otherwise, the request is replaced by a no-op. For hash function upgrades, in particular, the service state must be upgraded as well. This can be done gradually, a small number of AST subtrees per U-phase, by replacing node labels of nodes from tree leaves up to the root. While this upgrade takes place, some tree nodes will have labels computed with the old algorithm, and some labels with the new algorithm, but that is not a problem, until the old algorithm is ultimately and completely compromised.

6 Related Work

6.1 BFT Systems

Byzantine-fault tolerant state machine replication has received much attention in the systems community since PBFT [13]. Systems such as PBFT-PR [14] and COCA [59] have employed proactive recovery to reduce the vulnerability window. In these systems, a node is periodically rejuvenated, checking and repairing service state. COCA, in particular, shares a similar goal with Bonafide, i.e., the maintenance of a mapping from names to authenticators, but does not account for long-term operation in its structure or assumptions.

Researchers have proposed few improvements on PBFT to improve the $1/3$ fault bound. Like PBFT, BFT2F [34] provides safety and liveness with up to $1/3$ faulty replicas and then only fork* consistency (a weaker property than linearizability) when faults grow up to $2/3$ of the population. Although closer in spirit to our work, since it acknowledges that multiple fault thresholds might be useful towards different guarantees, BFT2F requires state at the clients, which is unreasonable for long-term preservation services, and offers fork*

consistency at its weakest fault threshold, which is inappropriate for an archival lookup service. Finally, our own A2M-enabled BFT protocols [16] improve fault bounds by using A2M. In particular, A2M-PBFT-EA provides both safety and liveness with up to fewer than $1/2$ faulty replicas. In this work we use some of the insights we gained in the A2M work, but focus instead on the notion of the tiered fault framework in a long-term service.

A2M [16] is a trusted primitive that removes the ability of faulty components to *equivocate*—tell different lies to different peers. Though a powerful primitive for BFT protocols, A2M is lacking MAS’s mode bit, which is critical for ensuring phase separations. In addition, A2M has more complicated internal structures and interfaces to account for linearizing requests and handling view changes. A2M has a set of trusted, undeniable, ordered logs, and it gives attestation of any entry or the last entry in the log or attestation vouching for some sequence numbers are skipped. In contrast, MAS has a set of storage slots with a simple write/read interface.

Recent work in Byzantine-fault tolerant storage systems has focused on developing efficient erasure-coding based block storage protocols [12, 22, 26] that reduce storage overhead. Hendricks, Ganger, and Reiter [26] developed the state-of-the-art BFT (m, n) erasure-coding block storage protocol to optimize reads and large writes. To tolerate f faults, the protocol requires $m \geq f + 1$ out of $n = m + 2f$ servers. These block storage protocols do not differentiate components of the systems and are not designed for long-term operation.

6.2 Differentiating Trust Levels

Researchers have differentiated trust levels on system components, failure types, and failure thresholds. The wormholes model is a *hybrid system model* where the system is decomposed into payload subsystems with weak assumptions and wormhole subsystems with strong assumptions and the two communicate through wormhole gateways [51, 53]. Wormholes such as the Timely Computing Base (TCB) [52] and the Trusted Timely Computing Base (TTCB) [17, 18, 41] provide concrete services such as timely execution and trusted block agreement to payload subsystems. TCB and TTCB are synchronous and fail by crashing. Similarly, we hybridize system components using tiers with different functionalities but we distinguish components explicitly for long-term operation and do in a finer granularity with different fault thresholds and in a more general way.

Hybrid fault models differentiate failure types on homogeneous systems: some nodes can have benign faults and others can have Byzantine faults [38, 50]. Furthermore, Byzantine faults are classified into malicious symmetric and malicious asymmetric faults [50]. Modified versions of the classic agreement algorithms can lead to

more flexible fault tolerance guarantees [9, 29, 50].

There has also been research on applying *different fault thresholds* to different sites or clusters. The multi-site threshold model differentiates two types of failures — site failures and process failures — in multi-site systems [28]. The model uses a fault threshold for the number of sites and a vector of fault thresholds, each of which is assigned to a site to account for a different process-failure probability depending on sites. In our tiered fault framework, each site is a tier. Yin et al. [57] proposed an architecture that separates execution from agreement: two groups of replicas— N agreement and M execution replicas—by dividing functionalities. This architecture can tolerate $\lfloor \frac{N-1}{3} \rfloor$ faults and $\lfloor \frac{M-1}{2} \rfloor$ faults, thus assigning different thresholds for the clusters. This partition is done based on functionalities. In our framework, each cluster is a tier. In Bonafide, we differentiate components based on functionalities but do at a finer level.

6.3 Long-term Stores

Self-certifying bitstore systems such as Glacier [25], PAST [44], OceanStore [31], Carbonite [15], and Antiquity [55] have addressed durability comprehensively. Authenticity is addressed by expecting all stored data to be *self-certifying*: the name of the datum is an *authenticator* for that datum, and can be used to verify its contents (e.g., via a cryptographic hash). However, such systems leave out of scope where those authenticators come from. It is precisely this gap that Bonafide seeks to fill: providing a long-term store for *non-self-certifying* information.

The LOCKSS system [36] is a digital preservation system not requiring an inviolable $1/3$ fault bound. However, this system is probabilistic in nature and does not provide hard safety or liveness guarantees as Bonafide does. POTSHARDS [48] is a long-term storage system that relies on multiple *separately-managed* archives. It uses secret splitting and stores shares into the archives to prevent accidental disclosure. Each object has a mapping between its object identifier and a hash for integrity. In POTSHARDS, each replica in an archive site is trusted or untrusted in its entirety. In comparison, in our work, only a small component at each replica is trusted (MAS), the update software can fail at up to a third of the population at the same time, and the service software can briefly fail everywhere at the same time without affecting safety.

CATS [58] is a single-server service that provides strong accountability of actions done by the server in a single authority and clients. Its approach is not to mask faults through replicated servers, but to detect faults and punish actors responsible for the faults. Its auditing scheme catches server rollback attacks probabilistically. In comparison, Bonafide provides *hard* safety and liveness guarantees under its fault assumption and considers replicated servers.

7 Conclusion

Long-term services that operate reliably are hard to construct. This work represents a step towards understanding better system structuring for long-term services that can lead to safer solutions. We present a tiered fault framework that partitions system components of nodes in different tiers, each enjoying a different fault threshold. We have designed and implemented Bonafide, a long-term key-value store that provides integrity under a three-tier Byzantine fault model. We hope that our work provides a framework for building more dependable long-term storage services.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and our shepherd, Alina Oprea, for her guidance.

References

- [1] Berkeley DB. <http://www.oracle.com/database/berkeley-db/index.html>.
- [2] Linuxbios. <http://linuxbios.org>.
- [3] sfslite. <http://www.okws.org/doku.php?id=sfslite>.
- [4] Trusted Computing Group (TCG). <http://www.trustedcomputinggroup.org/>.
- [5] 104th Congress, United States of America. Public Law 104-191: Health Insurance Portability and Accountability Act (HIPAA), Aug. 1996.
- [6] 107th Congress, United States of America. Public Law 107-204: Sarbanes-Oxley Act of 2002, July 2002.
- [7] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory. In *PODC*, 2004.
- [8] T. W. Arnold and L. P. V. Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 48(3/4), 2004.
- [9] M. H. Azmanesh and R. M. Kieckhafer. New hybrid fault models for asynchronous approximate agreement. *IEEE Trans. on Parallel and Distributed Systems*, 45(4), 1996.
- [10] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungle. A Fresh Look at the Reliability of Long-term Digital Storage. In *EuroSys*, 2006.
- [11] A. Buldas, P. Laud, and H. Lipmaa. Accountable certificate management using undeniable attestations. In *ACM CCS*, 2000.
- [12] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *IEEE DSN*, 2006.
- [13] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, 1999.
- [14] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *OSDI*, 2000.
- [15] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weather- spoon, M. F. Kaashoek, J. Kubiatawicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *NSDI*, 2006.

- [16] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested Append-Only Memory: Making Adversaries Stick to their Word. In *SOSP*, 2007.
- [17] M. Correia, N. F. Neves, L. C. Lung, and P. Verissimo. Low complexity byzantine-resilient consensus. *Distributed Computing*, 17(3), 2005.
- [18] M. Correia, P. Verissimo, and N. F. Neves. The design of a COTS real-time distributed security kernel. In *European Dependable Computing*, 2002.
- [19] C. De Cannière and C. Rechberger. Preimages for Reduced SHA-0 and SHA-1. In *CRYPTO*, 2008.
- [20] M. Factor, D. Naor, S. Rabinovici-Cohen, L. Ramati, P. Reshef, and J. Satran. Preservation datastores: Architecture for preservation aware storage. In *IEEE MSST*, 2007.
- [21] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Super-efficient verification of dynamic outsourced databases. In *RSA Conference—Crypto Track*, 2008.
- [22] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *IEEE DSN*, 2004.
- [23] S. Haber and W. S. Stornetta. How to Time-stamp a Digital Document. *J. of Cryptology*, 3(2), 1991.
- [24] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical Accountability for Distributed Systems. In *SOSP*, 2007.
- [25] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *NSDI*, 2005.
- [26] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-Overhead Byzantine Fault-Tolerant Storage. In *SOSP*, 2007.
- [27] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, and T. Wobber. Sealing OS processes to improve dependability and safety. In *EuroSys*, 2007.
- [28] F. P. Junqueira and K. Marzullo. The virtue of dependent failures in multi-site systems. In *HotDep*, 2005.
- [29] R. M. Kieckhafer and M. H. Azamanesh. Reaching approximate agreement with mixed mode faults. *IEEE Trans. on Parallel and Distributed Systems*, 3(1), 1994.
- [30] P. C. Kocher. On certificate revocation and validation. In *Financial Cryptography*, 1998.
- [31] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *ASPLOS*, 2000.
- [32] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2), 1998.
- [33] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM TOCS*, 1992.
- [34] J. Li and D. Mazières. Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems. In *NSDI*, 2007.
- [35] P. Maniatis and M. Baker. Secure History Preservation Through Timeline Entanglement. In *USENIX Security*, 2002.
- [36] P. Maniatis, M. Roussopoulos, T. Giuli, D. S. H. Rosenthal, and M. Baker. The LOCKSS Peer-to-Peer Digital Preservation System. *ACM TOCS*, 23(1), 2005.
- [37] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1987.
- [38] F. Meyer and D. Pradhan. Consensus with dual failure modes. In *IEEE Fault-Tolerant Computing*, 1987.
- [39] M. Naor. Bit commitment using pseudorandomness. *Journal of Cryptology*, pages 151–158, 1991.
- [40] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI*, 1996.
- [41] N. F. Neves, M. Correia, and P. Verissimo. Solving Vector Consensus with a Wormhole. *IEEE Trans. on Parallel and Distributed Systems*, 16(12), 2005.
- [42] C. Preimesberger. Intel Faces Up to E-Mail Retention Problems in AMD Lawsuit. *eWeek.com*, Mar. 2007. Fetched on 10/9/2007 from <http://www.eweek.com/article2/0,1759,2101674,00.asp>.
- [43] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *FAST*, 2002.
- [44] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *SOSP*, 2001.
- [45] R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *SOSP*, 2003.
- [46] P. Sousa, N. F. Neves, and P. Verissimo. Proactive Resilience through Architectural Hybridization. In *ACM Symposium on Applied Computing*, 2006.
- [47] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, Network Working Group, 1995.
- [48] M. W. Storer, K. Greenan, E. L. Miller, and K. Voruganti. POTSHARDS: Secure Long-Term Storage Without Encryption. In *USENIX ATC*, 2007.
- [49] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST*, 2008.
- [50] P. Thambidurai and You-keun Park. Interactive consistency with multiple failure modes. In *SRDS*, 1988.
- [51] P. Verissimo. Uncertainty and Predictability: Can they be reconciled? *LCNS: FuDiCo*, 2584, 2003.
- [52] P. Verissimo, A. Casimiro, and C. Fetzer. The timely computing base: Timely actions in the presence of uncertain timeliness. In *IEEE DSN*, 2000.
- [53] P. E. Verissimo. Travelling through wormholes: a new look at distributed systems models. *ACM SIGACT News*, 37(1), 2006.
- [54] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM*, 2004.
- [55] H. Weatherspoon, P. Eaton, B.-G. Chun, and J. Kubiatowicz. Antiquity: Exploiting a secure log for wide-area distributed storage. In *EuroSys*, 2007.
- [56] Y. Xie and A. Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *International Conference on Computer Aided Verification*, 2005.
- [57] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for

- Byzantine Fault Tolerant Services. In *SOSP*, 2003.
- [58] A. R. Yumerefendi and J. S. Chase. Strong Accountability for Network Storage. In *FAST*, 2007.
- [59] L. Zhou, F. B. Schneider, and R. V. Renesse. COCA: A secure distributed online certification authority. *ACM TOCS*, 20(4), 2002.

Notes

¹We use the terms *fault bound* and *fault threshold* interchangeably.

²The window of vulnerability varies depending on system conditions. For example, if some replicas' state is corrupted, the window becomes large.

³This metaphor is usually attributed to Reagan Moore.

⁴With our recent A2M-PBFT-EA protocol [16], we can improve this fault bound from $1/3$ to $1/2$. We leave the details out of this paper to keep the exposition simple.

⁵We chose Berkeley DB since it was readily available, but our design would be compatible with any block-store such as Venti [43].

A Correctness Arguments

In this appendix, we prove that Bonafide provides the integrity property under the tiered Byzantine-fault model. In the proof, we denote by $s(r)$ the S phase of round r and by $p(r)$ the U phase after $s(r)$. Without loss of generality consider a binding (k, v) .

Lemma A.1. *For every $\text{Add}(k, v)$ request accepted by $2f + 1$ replicas during a S phase in which there is no more than f faulty replicas out of $3f + 1$ total replicas, (k, v) appears in any valid BATCH certificate.*

Proof. We say an $\text{Add}(k, v)$ request is accepted if there are at least $2f + 1$ replicas that receive the request; a client can ensure that the request is accepted by checking authenticated tentative ADD responses. Let Q_a denote this set of replicas. At the start of $p(r)$, each replica multicasts a BATCH message to other replicas. The leader collects $2f + 1$ distinct BATCH messages that form a BATCH certificate. Let Q_b denote the set of replicas that form this certificate. $Q_a \cap Q_b$ includes at least one non-faulty replica that receives the ADD request since in the S phase the number of faulty replicas is no more than f . Therefore, the accepted request is contained in the BATCH certificate. \square

Lemma A.2. *A stable checkpoint certificate of the previous round appears in any valid BATCH certificate.*

Proof. We show that at $p(r)$ the BATCH certificate contains the stable checkpoint ($2f + 1$ matching MAS attestations) of $p(r - 1)$. At $p(r - 1)$, there are at least $2f + 1$ replicas, each of which creates a stable checkpoint certificate and puts the certificate to its MAS. Let Q_p denote this set of replicas. $Q_p \cap Q_b$ intersects at at least $f + 1$ replicas and hence includes at least one common non-faulty replica between two quorums. This replica ensures that the stable checkpoint of the previous round is

included in the BATCH certificate. Therefore, the BATCH certificate of $p(r)$ contains the correct stable checkpoint of $p(r - 1)$. \square

Theorem A.3. *If a binding (k, v) is accepted at $s(r)$ and k is not in the AST, the binding is correctly read (or temporarily unavailable) at all $s(r')(r' > r)$.*

Proof. From Lemmas A.1 and A.2, we know that a PROPOSE message with a valid BATCH certificate contains a correct stable checkpoint certificate of the previous round and bindings received during a S phase in which there is no more than f faulty replicas. When the leader invokes PBFT with the PROPOSE message, PBFT ensures that all non-faulty replicas agree on the PROPOSE message. Each such replica checks that k does not exist; if necessary, the replica may perform state transfer for this validation. If k does not exist, the replica inserts (k, v) into the AST, computes a new AST digest, and appends it to MAS. Finally, each replica creates a stable checkpoint certificate by collecting $2f + 1$ matching UCHECKPOINT messages and appends the certificate to its MAS.

Now, suppose a client gets a reply certificate ($f + 1$ matching MAS attestations) of $\text{Get}(k)$ at $s(r + 1)$. The reply certificate contains at least one *up-to-date* replica since a non-faulty replica enters $s(r + 1)$ only after creating a stable checkpoint certificate. Therefore, a client correctly reads value v when it queries with k at $s(r + 1)$.

Once (k, v) is inserted into Bonafide at $p(r)$, it is clear that $p(r + 1)$ carries (k, v) from $p(r)$ correctly with the same argument we make for $p(r - 1)$ and $p(r)$ above since we have a correct AST digest. We can inductively argue the same holds for $p(r + i)$ and $p(r + i + 1)$ for all $i \geq 0$. Therefore, when a client gets a reply certificate for $\text{Get}(k)$ at all $s(r + i)$ ($i > 0$), the client receives correct (k, v) . \square

A Systematic Approach to System State Restoration during Storage Controller Micro-Recovery

Sangeetha Seshadri*

Lawrence Chiu[†]

Ling Liu*

*Georgia Institute of Technology
801 Atlantic Drive GA-30332
{sangeeta,lingliu}@cc.gatech.edu

[†]IBM Almaden Research Center
650 Harry Road CA-95120
{lchiu}@us.ibm.com

Abstract

Micro-recovery, or failure recovery at a fine granularity, is a promising approach to improve the recovery time of software for modern storage systems. Instead of stalling the whole system during failure recovery, micro-recovery can facilitate recovery by a single thread while the system continues to run. A key challenge in performing micro-recovery is to be able to perform efficient and effective state restoration while accounting for dynamic dependencies between multiple threads in a highly concurrent environment. We present Log(Lock), a practical and flexible architecture for performing state restoration without re-architecting legacy code. We formally model thread dependencies based on accesses to both shared state and resources. The Log(Lock) execution model tracks dependencies at runtime and captures the failure context through the restoration level. We develop restoration protocols based on recovery points and restoration levels that identify when micro-recovery is possible and the recovery actions that need to be performed for a given failure context. We have implemented Log(Lock) in a real enterprise storage controller. Our experimental evaluation shows that Log(Lock)-enabled micro-recovery is efficient. It imposes < 10% overhead on normal performance and < 35% overhead during actual recovery. However, the 35% performance overhead observed during recovery lasts only six seconds and replaces the four seconds of downtime that would result from a system restart.

1 Introduction

Enterprise storage systems serve as repositories for huge volumes of critical data and information. Unavailability of these systems results in losses amounting to millions of dollars per hour [1], bringing organizations to a grinding halt.

Most existing work in the domain of storage system availability addresses failures of the storage media (such as disks) and recoverability from these failures [2, 3, 4]. However, failures at the firmware layer that result in service loss remain largely unaddressed. At the same time, the software at

the firmware layer of a storage system has evolved tremendously in terms of functionality. Modern storage controllers are highly concurrent embedded systems with millions of lines of code [5, 6]. As a result of this complexity, recovering from controller failures is both difficult and expensive.

While system availability requirements are constantly being driven higher, failure recovery time is increasing due to increasing system size, higher performance expectations, virtualization and consolidation. Since software failure recovery is often performed through system-wide recovery, the recovery process itself does not scale with system size [6, 7, 8].

How can failure recovery be made scalable? Partitioning the system into smaller components with independent failure modes can reduce recovery time. However, it also increases management costs and decreases flexibility, while still being susceptible to sympathetic failures. On the other hand, refactoring the software into smaller independent components in order to use techniques such as micro-reboots [8] or software rejuvenation [9] requires sizable investments in terms of development and testing costs, unacceptable in the case of legacy systems. An alternative approach is to be able to perform fine-granularity recovery or *micro-recovery*, without re-architecting the system. Under this approach, failure recovery is targeted at a small subset of tasks/threads that need to undergo recovery while the rest of the system continues uninterrupted.

Enabling fine grained recovery can be challenging, especially in legacy systems, and the following issues must be addressed:

- **Evaluating recovery success:** What are the failures that can effectively and efficiently be recovered from, using micro-recovery?
- **Determining recovery actions:** What are the recovery strategies and recovery actions that must be performed in order to restore the system from an error state to an error-free state?
- **Identifying dependencies:** Given the large number of dynamic dependencies possible in a highly concurrent system, what is the scope of fine-granularity recovery?

- **Enhancing recovery success and efficiency:** How can we enhance the system to facilitate better recovery success and efficiency?

We address the first three questions, focusing on the challenges of tracking and restoring system state during micro-recovery, evaluating the possibility of recovery success and determining recovery actions.

We make two unique contributions in terms of effective state restoration during micro-recovery. First, by analyzing the system state space, we identify the set of events and system states that affect state restoration from the perspective of micro-recovery. We introduce the concepts of *Restoration levels* and *Recovery points* to capture failure and recovery context and describe how to flexibly evaluate the possibility of recovery success. Based on the restoration levels and recovery points, we introduce *Resource Recovery Protocol (RRP)* and *State Recovery Protocol (SRP)*, which provide rules to guide state restoration.

Our second contribution is Log(Lock), a practical and lightweight architecture to track dependencies and perform state restoration in complex, legacy software systems. Log(Lock) passively logs system state changes to help identify dependencies between multiple threads in a concurrent environment. Utilizing this record of state changes and resource ownership, Log(Lock) provides the developer with the failure context necessary to perform micro-recovery. Recovery points and their associated recovery handlers are specified by the developer. Log(Lock) is responsible for tracking dependencies and computing restoration levels at runtime.

We have implemented and evaluated Log(Lock) in a real enterprise storage controller. Our experimental evaluation shows that Log(Lock)-enabled micro-recovery is both efficient (<10% impact on performance) and effective (reduces a four second downtime to only a 35% performance impact lasting six seconds). In summary, micro-recovery with Log(Lock) presents a promising approach to improving storage software robustness and overall storage system availability.

2 Log(Lock): Design Overview

This section gives an overview of the Log(Lock) system design. We first describe the problem statement that motivates the Log(Lock) design. Using examples, we highlight the unique characteristics of micro-recovery in the context of highly concurrent storage controller software. Then we outline the technical challenges for systematic state restoration during micro-recovery. Finally, we briefly describe the system architecture of Log(Lock).

2.1 Motivation

In this section, we motivate the need for a flexible and lightweight state restoration architecture using a highly concurrent storage controller. The storage controller refers to the firmware that controls the cache and provides advanced functionality such as RAID, I/O routing, synchronization with remote instances and virtualization. In modern enterprise-class storage systems, the storage controller has evolved to become extremely complex with millions of lines of code that is often difficult to test. The controller code typically executes over an N-way processing complex using a large number of short concurrent threads (~20 million/minute). While the software is designed to extract maximum concurrency and satisfy stringent performance requirements, unlike database transactions it does not adhere to ACID (atomicity, consistency, isolation and durability) properties. This software is representative of a class expected to sustain high throughput and low response times continuously.

With this architecture, when one thread encounters an exception that causes the system to fail, the common way to return the system to an acceptable, functional state is by restarting and reinitializing the entire system. While the system reinitializes and waits for the operations to be redriven by a host, access to the system is lost contributing to downtime. As the system scales to larger number of cores and as the size of the in-memory structures increase, such system-wide recovery will no longer scale [6, 8].

Many software systems, especially legacy systems, do not satisfy the conditions outlined as essential for micro-rebootable software [8]. For instance, even though the storage software may be reasonably modular, component boundaries, if they exist, are loosely defined and components are stateful. Under these circumstances, the scope of a recovery action is not limited to a single component.

The goal of micro-recovery is to perform recovery at a fine granularity such as at the thread-level, while determining the scope of recovery actions dynamically, based on dependencies identified at runtime. The key challenges in performing micro-recovery are identifying dependencies based on failure and recovery context, determining recovery actions and restoring the system to a consistent state after a failure.

2.2 Examples

We present three real examples from a storage controller software. We demonstrate how the semantics and success of fine-grained recovery are determined by failure context and the interactions of threads.

Figure 1 shows two code snippets: R1 increments the number of active users before performing work


```

R1: /* Increment number of Users */
lockWrite( &numActiveUsersLock);
numActiveUsers ++;
unlockWrite( &numActiveUsersLock);
...
...
/* Decrement number of Users */
lockWrite( &numActiveUsersLock);
numActiveUsers --;
unlockWrite( &numActiveUsersLock);

R2: /* Start background tasks if no users active */
lockRead ( &numActiveUsersLock);
if ( numActiveUsers == 0 ) {
    Start performing background tasks.
}
unlockRead( &numActiveUsersLock);

```

Figure 1: Lost Update Conflict

```

R3: /* Get cache track to write to fast-write cache */
startSCSIcmd();
└─ processRead();
   └─ getCacheTrack();
      └─ getTempResource() {
          ...
          PANIC
      }

```

Figure 2: Resource Ownership Conflict

and in R2, a background job is triggered when there are no active users in the system. When a panic (user defined or system failure/exception) occurs during the execution of region R1, then assume that the micro-recovery strategy is to reattempt execution of region R1. The recovery action must ensure clean relinquishing of resources such as the lock *numActiveUsersLock*. It is also important to ensure that the system state is consistent since corruption of the counter can either cause the background jobs to never be triggered or to be triggered in the presence of active users. In Example-1, while it is permissible for other threads to read the value of the *numActiveUsers* count at anytime provided the *numActiveUsersLock* has been released, the system must ensure that if and only if a thread fails after performing an increment operation on the count, a decrement operation is performed during recovery. On the other hand, if the failure was caused during the execution of region R2, an idempotent background task that is not critical, the recovery strategy may be to just abort the current execution of the background task. However, recovery must ensure that the lock *numActiveUsersLock* has been released.

Figure 2 shows the processing of a write command. In the event of encountering a failure, state restoration must ensure that temporary resources obtained from a shared pool are freed correctly in order to avoid resource leaks or starvation. It may

```

R4: /* Update Metadata Location */
lockWrite( &MetadataLocationLock);
MetadataLocation = XX;
unlockWrite( &MetadataLocationLock);
...

```

Figure 3: Dirty Read Conflict

also require that certain cache tracks are checked for consistency, depending upon the point of failure. However, for a resource such as a buffer or empty cache track obtained from a shared pool, restoring the previous contents is not necessary.

Figure 3 shows a thread that updates a global variable indicating the metadata location, such as for checkpoint activity. In the event of a failure caused due to a failed location, the thread may have the opportunity to modify the location without notifying other threads or causing inconsistency, provided no other thread has already consumed the value. However if that is the case, the system may have to resort to recovery at a higher level.

These examples highlight the fact that consistency requirements for state restoration vary with failure context. For example, in the case of a counter generating unique numbers, the only requirement may be that modifications are monotonous. For a shared resource, the state remains consistent as long as there are no resource leaks that could eventually lead to starvation and system unavailability. Unlike a transactional system, where similar problems are addressed, the semantics of the state and failure may render certain types of conflicts irrelevant from the perspective of system recovery. This emphasizes the need for a flexible state restoration architecture that is also lightweight and efficient, thereby allowing the system to sustain high performance.

2.3 Failure Model

Our work is targeted at transient failures in the system, especially failures where the developer now uses system restart as a method to take the system from an unknown or faulty state to a known state. A number of such failure scenarios occur in storage controller software and may apply equally well to other software systems. We present some examples from our analysis of storage controller failures. Bad input from administrator or user, insufficient error handling, deadlocks, a faulty communication channel, unhandled race conditions, boundary conditions, and timeouts are some examples of such failures seen in storage controllers. The system restart mechanism is used often because the system has insufficient information, for example, when reacting to an asynchronous event or when dealing with an unknown state or receiving an unexpected stimulus.

For example, consider a failure scenario where a write operation to disk fails because a driver from a third party vendor returns an unidentified error code due to a bug in its code. In this case, since writes are buffered in a fast write cache and the actual write to disk is performed asynchronously, dropping the request is not an option. Another example is a configuration issue that appeared early in the installation process that may have been fixed by trying various combinations of actions that were not correctly undone. As a result the system finds itself in an unknown state that manifests as a failure after some period of normal operation. Such errors are difficult to trace, and although transient may continue to appear every so often.

Some transient failures can be fixed through appropriate recovery actions that may range from dropping the current request to retrying the operation or performing a set of actions that take the system to a known consistent state. Some other examples of such transient faults that occur in storage controller code are: (1) An unsolicited response from an adapter - An adapter (a hardware component not controlled by our microcode) sends a response to a message which we did not send - or do not remember sending; (2) Incorrect Linear Redundancy Code (LRC): A control block has the wrong LRC check bytes, for instance, due to an undetected memory error; (3) Queue full condition: An adapter refuses to accept more work due to a queue full condition. In addition, there are other error scenarios such as violation of service level agreements. The 'time-out' conditions are also common in large scale embedded storage systems. While the legacy system grows along multiple dimensions, the growth is not proportional along all dimensions. As a result hard-coded constant timeout values distributed in the code base often create unexpected artificial violations. For a more detailed classification of software failures, please refer to [6].

2.4 Technical Challenges

With software recovery, state restoration actions depend on the actions of the failed thread and its interactions with state and shared resources.

Threads in the system interact in two fundamental ways: (1) reading/writing shared data and (2) acquiring/releasing resources from/to a common pool. Threads also interact with the outside world through actions such as positioning a disk head or sending a response to an I/O. Often these actions cannot be rolled back and are referred to as *outside world processes (OWP)* [10]. In such a system, state restoration and micro-recovery must consider the sequence and interleaving of the actions of concurrent threads that gives rise to the following conflicts:

- **Dirty Reads (Write-Read Conflict):** Data written by the failed thread has already been consumed by another thread.
- **Lost Updates (Write-Write Conflict):** Rolling back the failed thread may cause the updates of other threads to be overwritten or lost.
- **Unrepeatable Reads (Read-Write Conflict):** The value of the shared state variable required by the failed thread has already been overwritten.
- **Resource Ownership :** The failed thread may continue to be in the possession of resources from a shared pool or may be holding a lock resulting in resource leaks or starvation issues.

The above taxonomy is derived from that used to describe concurrency control concepts in transaction processing systems [11]. For a given failure, the set of recovery actions that need to be performed to return the system to a consistent state may vary depending upon the failure and the occurrence of one or more of the above conflicts. Note that for application state, the intention is not to deterministically replay the events before the failure, or recover the application state to exactly as it was at the instant of failure. Rather, the goal is to restore the system to an error-free state. In fact, the recovery strategy may itself explicitly rely on non-determinism to remove transient failures. For example, Rx [12] demonstrates an interesting approach to recovery by retrying operations in a modified environment using checkpointed system states for rollbacks.

Checkpointing for fault-tolerance is a well known technique [10, 12, 13, 14] that has also been applied to deterministic replay for software debugging [15, 16, 17]. However, checkpointing techniques are mostly targeted at long-running applications [10] such as scientific workloads [13], or applications where the system can tolerate the overhead imposed by checkpointing [12, 14]. A number of unique challenges in the case of storage controller software make checkpointing infeasible: Unlike long-running applications, storage controllers have a high rate of short ($< 500\mu\text{secs}$) concurrent threads and are designed to support extremely high throughput and low response times. Given the highly concurrent nature of controllers, both quiescing the system in order to take the checkpoint, as well as logging the tasks in order to re-execute work beyond the checkpoint is expensive in terms of time and space - especially since system state includes large amounts of metadata and cached data. Next, communication with OWPs such as hosts and media cannot be rolled back and hence invalidates checkpoints. Finally, due to the complexity of the code, not all failures will be amenable to micro-recovery, making checkpointing

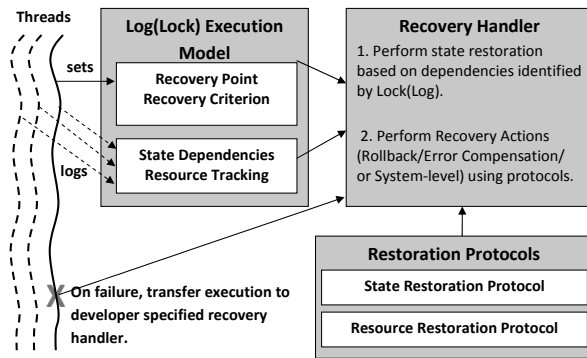


Figure 4: Log(Lock) Architecture Overview

too heavy weight.

State restoration and conflict serialization is also of interest to transactional systems [18]. Transactional databases use schemes like strict 2-phase locking (2PL) to guarantee conflict serializability [19]. However, such techniques can increase the length of critical sections (i.e. durations of locks) and are inefficient for the highly-concurrent storage controller environment. Moreover, we show in Section 2.2 that, recovery actions are determined based on both the context and semantics of failure and a “one size fits all” serializability, while simplifying recovery procedures, can constrain the recovery process.

2.5 System Architecture

The Log(Lock) architecture provides support for state restoration during micro-recovery. To achieve this goal, Log(Lock) tracks resources and state dependencies relevant to a thread that has incorporated recovery handlers for micro-recovery.

Figure 4 presents an overview of our system architecture and describes the roles played by the Log(Lock) execution model and restoration protocols. The figure shows a system with concurrently executing threads where the thread depicted by a solid line incorporates micro-recovery mechanisms. In order to facilitate micro-recovery, the thread sets recovery points during execution, where each recovery point is associated with a recovery criterion. The recovery criterion specifies the conditions that must be satisfied by the failure context in order to use the recovery point as a starting point for recovery. Using the Log(Lock) architecture, the thread (depicted by a solid line) enabled with micro-recovery mechanisms indicates state and resources that are relevant to recovery. Log(Lock) then begins logging all relevant changes and dependencies, based on the actions of both this thread and other concurrent threads (depicted by dotted lines).

In the event of a failure, control transfers to a developer specified recovery handler. The handler

performs state restoration actions by utilizing the resource tracking and state dependency information provided by the Log(Lock) execution model, in consultation with the restoration protocols. It also decides on an appropriate recovery strategy such as rollback, error compensation or system-level recovery. The implementation of the Log(Lock) dependency tracking component must ensure efficiency during normal operation while the recovery protocols ensure consistency of state restoration during failure recovery. Below, we summarize the four primary design objectives of Log(Lock):

- **Incremental:** Allow micro-recovery to be applied incrementally to handle failures depending upon effectiveness of a fine-grained approach.
- **Lightweight and Non-intrusive:** Minimize impact on system performance and modifications to legacy software functional architecture.
- **Dynamic:** Handle dynamic dependencies.
- **Flexible:** Allow application developers the flexibility to treat different failures differently without enforcing a “one size fits all” consistency requirement, allowing a larger number of failures to be handled correctly at a fine-granularity.

In the next two sections, we first describe the concepts of ‘restoration levels’ and ‘recovery points’ and present the restoration protocols. Then, we present the Log(Lock) execution model and illustrate application of the protocols through example scenarios.

3 State Space Exploration

In this section, we model failure scenarios and recovery contexts using a state space analysis approach. Our approach is based on the intuition that in a concurrent system, global state and shared resources are often protected by locks or similar primitives.

This section is divided into two parts. In the first part, we model system events, state transitions and interleaving of concurrent threads and demonstrate the discrete state space and recovery scenarios. We introduce the concepts of *Restoration Level* and *Recovery Criterion*, that help match a failure context to a recovery strategy. In the second part, we systematically identify the set of recovery strategies that can be applied to each failure scenario and present two protocols for state restoration. The **Resource Recovery Protocol (RRP)** defines the steps to handle resource ownership conditions and the **State Recovery Protocol (SRP)** sets forth the rules to perform state restoration.

3.1 Modeling Thread Dependencies

Let $\mathcal{T} = \{T_i | 1 \leq i \leq n\}$ define a system with n concurrent threads. Let $\mathcal{X}_i(t)$ denote the sequence of

Table 1: Valid States for Thread T_i

Notation	Description
$T_i S$	T_i initial state
$T_i R$	T_i holds a read lock
$T_i W$	T_i holds an exclusive write lock
$T_i U$	T_i has released the lock
$T_i F$	T_i is in failed state
$T_i A$	T_i acquired a resource
$T_i Re$	T_i released a resource
$T_i E$	T_i performed an externally visible action

states of thread T_i up to time t . The schedule $\mathcal{S}(t)$ at time t is the interleaving of the sequence of actions in $\mathcal{X}_i(t)$ for each thread T_i . Let v denote a globally shared structure protected by a lock. Table 1 shows the list of valid states for a thread.

The system implements micro-recovery at a thread granularity. Any failure that cannot be handled by micro-recovery is resolved using a system-level recovery mechanism (e.g. software reboots).

The state space for system execution consists of all legitimate schedules $\mathcal{S}(t)$. System states that represent the failed state of one of the executing threads are relevant from the perspective of micro-recovery. To simplify the subsequent discussion, we apply the following rules to reduce the state space:

- We consider the interactions between only two threads T_1 and T_2 .
- We only consider system states where the last state of thread T_1 is $T_1 F$.
- Only T_1 encounters a failure. Failures of thread T_2 are symmetric and can be treated similarly.
- Read or write actions performed by T_2 before any such actions by T_1 are ignored.
- We assume that the system can recover from only a single failure. Failure during recovery results in system-level failure recovery.
- The externally visible action is equivalent to a ‘commit action’ that cannot be rolled back.

Occurrences of the following patterns in the schedule $\mathcal{S}(t)$ are of interest and relevant to the selection of a recovery strategy by thread T_1 . Let \rightarrow denote the “happened before” relation [20].

- **Dirty Read (DR):** $T_1 W \rightarrow T_2 R \rightarrow T_1 F$.
- **Lost Update (LU):** $T_1 W \rightarrow T_2 W \rightarrow T_1 F$.
- **Unrepeatable Read (UR):** $T_1 R \rightarrow T_2 W \rightarrow T_1 F$.
- **Residual Resources (RR):**
 $(T_1 R \rightarrow T_1 F) \wedge (T_1 U \rightarrow T_1 F)$ or $(T_1 W \rightarrow T_1 F) \wedge (T_1 U \rightarrow T_1 F)$
or $(T_1 A \rightarrow T_1 F) \wedge (T_1 Re \rightarrow T_1 F)$.
- **Committed Dependency (CD):**
 $T_1 W \rightarrow T_2 R \rightarrow T_2 E \rightarrow T_1 F$ or $T_1 W \rightarrow T_2 W \rightarrow T_2 E \rightarrow T_1 F$
or $T_1 R \rightarrow T_2 W \rightarrow T_2 E \rightarrow T_1 F$.

To determine the right strategy for recovery, it is important to determine which of the above conflicts have occurred and are relevant to recovery.

Restoration Level: The restoration level $\mathcal{R}_i(t)$ of a thread T_i at instant t , is a 5-tuple $\langle DR, LU, UR, RR, CD \rangle$ indicating the occurrence of dirty reads, lost updates, unrepeatable reads, residual resources and committed dependencies in $\mathcal{S}(t)$.

Recovery Point: A recovery point p_i in thread T_i represents an execution point to which control is transferred at the end of a recovery procedure. A default recovery point defined for all threads is the initial system state.

Recovery Criterion: Each recovery point p_i is associated with a recovery criterion \mathcal{C}_i which is a 4-tuple $\langle DR, LU, UR, RR \rangle$ that represents the set of criteria for dirty reads, lost updates, unrepeatable reads and residual resources, that the system state should satisfy before recovery can be attempted using p_i . For the default recovery point, all elements of the recovery criterion are defined as “don’t care”.

CD does not figure in the recovery criterion since this information is used only to choose between alternate recovery strategies in the recovery handler. We discuss the use of CD conditions during recovery in the state recovery protocol in Section 3.2. In our current design, recovery points and their associated recovery handlers are identified by developers and are associated to an execution context. When a thread leaves a context, the associated recovery points go out of scope. Within a single execution context, multiple recovery points may be defined, any of which could potentially be used during recovery. Then the appropriate recovery point for the current failure scenario is chosen by the logic in the recovery handler. In the developer-specified recovery handler, the feasibility and correctness of restoring the failed system state using a recovery point, is determined using the resource and state recovery protocols described next. Once the valid recovery points have been identified from the available choices, the selection of an appropriate recovery point and recovery strategy may be a decision depending upon factors such as the amount of resources available for recovery and the time required to complete recovery.

3.2 Restoration Protocols

We consider the following possible recovery strategies: (1) Rollback; (2) Roll-forward style recovery or error compensation; (3) System-level recovery [21]. Of these the rollback and error compensation strategies may be applied to the failed thread only (single-thread recovery) or to multiple threads including the failed thread (multi-thread recovery). The following

protocols are based on the assumption that committed dependencies cannot be rolled-back.

Resource Recovery Protocol (RRP): System state can be restored to recovery point p_i only if $\mathcal{R}_i(t)$ meets \mathcal{C}_i on the RR criterion. Otherwise, the thread must first attempt to release or acquire resources to meet the criterion.

The state recovery protocol (SRP) specifies the recovery strategies applicable for different failure and recovery contexts. The rationale behind the SRP rules is that an occurrence of DR, LU or UR events imply that an interaction with other concurrent threads in the system have occurred. When the restoration level does not meet the recovery criterion and interactions with other threads have occurred, then single thread recovery is no longer sufficient. Next, the success of multi-thread recovery depends on the occurrence of an externally visible action and whether the dependency has already been committed. Concretely, the rules of state recovery are:

State Recovery Protocol (SRP): 1. To perform single-thread recovery and restore state to recovery point p_i , $\mathcal{R}_i(t)$ should meet \mathcal{C}_i on every element of \mathcal{C}_i .
2. If $\mathcal{R}_i(t)$ does not meet \mathcal{C}_i on DR, LU, UR conditions and CD occurs in $\mathcal{S}(t)$, then only error compensation or system-level recovery can be attempted.
3. If $\mathcal{R}_i(t)$ does not meet \mathcal{C}_i on DR, LU, UR conditions and CD has **not** been observed in $\mathcal{S}(t)$, then only multi-thread rollback, error compensation or system-level recovery is possible.

4 Log(Lock) Execution Model

In this section, we present a concrete execution model of Log(Lock), that utilizes the state space analysis presented in the previous section. We show how to decide recovery strategies and how restoration levels can be tracked practically. Although the discussion in this paper focuses on a thread-level recovery granularity, the Log(Lock) architecture can easily be extended to a more coarse granularity of micro-recovery such as at a task or component level.

In a complex legacy system such as a storage controller, not all failures can be handled efficiently through fine-grained recovery - either because the failure and recovery code may be too complex, or system-level recovery may be a more effective recovery technique, or simply because there may be insufficient development and testing resources. Therefore, our approach first involves identifying candidates for fine-grained recovery based on the analysis of failure logs and the software itself. The executing instance of each candidate is known as a **recoverable thread**. Recall that, for each recoverable

thread multiple recovery points and associated recovery criterion may be defined. In the event of a failure, control is transferred to the recovery handler (Section 2.5).

4.1 Tracking State Changes

Log(Lock) is based on the intuition that all shared state and resources are protected by locks or similar synchronization primitives. Tracking lock/unlock calls can therefore guide the understanding of system state changes and provide the information required to identify the restoration level at the instant of failure. At the same time, by tracking these calls on resources and applying the resource recovery protocol, we can prevent deadlocks or resource starvation issues. In order to compute restoration levels and perform system state restoration, Log(Lock) maintains the following:

Undo Logs: Undo logs are local logs maintained by each recoverable thread for the following purposes: (1) Track the sequence of state changes within a single thread; (2) Track the creation of recovery points and (3) Track resource ownership. In general, the Undo logs can be used to encode any information required by a thread's recovery handler. In our current implementation, Undo-logging activities and maintenance of the Undo logs are left to the developer.

Change Track Logs: In order to track conflicts between concurrent threads, Log(Lock) maintains Change Track Logs for each lock. The Change Track Log is used to: (1) Track concurrent changes to shared structures and (2) Track commit actions.

Both the Undo Log and Change Track Logs are maintained only in main memory and are verified for integrity using checksums. In our implementation, the change track log is implemented as a hashtable indexed using the pointer to the lock as key. Unlike database logs or checkpoints for state restoration, these logs do not need to be flushed to stable storage. If a failure crashes the system causing it to lose or corrupt the logs, then we must perform a system-level restart to restore the system to a consistent, functional state and no longer require the software's state restoration logs from before the failure.

Log(Lock) provides four basic primitives to a recoverable thread:

- *startTracking(lock)*: Start tracking changes to the structure protected by *lock*.
- *stopTracking(lock)*: Stop tracking changes to the structure protected by *lock*.
- *getRestorationLevel(lock)*: Compute the restoration level for the structure protected by *lock*.
- *getResourceOwnership(lock)*: Get ownership information (including lock ownership) for the

```

/* Recovery Criterion for R1: No residual resources */
Owner = getResourceOwnership(&numActiveUsersLock);
/* Acquire ownership in write mode for consistent recovery */
if( Owner == ReadMode) {
    unlockRead(&numActiveUsersLock);
    lockWrite(&numActiveUsersLock);
} else if(!Owner)
    lockWrite(&numActiveUsersLock);
level = getRestorationLevel(&numActiveUsersLock);

if ( level indicates dirty reads or lost updates ) {
    /* Indicates write completed */
    numActiveUsers -- ;
} else {
    /* No other operations or write may not have completed */
    Replace old value using the Undo log;
}
unlockWrite( &numActiveUsersLock);
/* State restore complete. Jump to new execution point */
Jump to R1;

```

Figure 5: State Restoration Using Log(Lock)

structure protected by *lock*.

All the above primitives are explicitly inserted into the code by the developer. The *startTracking* call is used to trigger change tracking for shared state and resources protected by the *lock* parameter. These accesses are identified by trapping lock/unlock calls. When the recoverable thread determines that the logs for a particular structure are no longer required, it explicitly issues a *stopTracking* call. In the event of a failure, the system transfers control to the designated recovery handler. The recovery handler can utilize the *getRestorationLevel* and *getResourceOwnership* primitives to determine the current restoration level and resource ownership and then invoke recovery procedures appropriately. The restoration level is determined by examining the undo and change track logs.

4.2 Recovery Using Restoration Protocols

The goal of our state restoration approach is to return the system to a correct, functional and known state by performing localized recovery and state restoration actions. The recovery actions are targeted at only a small subset of the threads in the system and a small region of the total system state that has been identified as affected by failure-recovery. Figure 5 shows pseudo code for state restoration using the restoration protocols and the Log(Lock) architecture for the scenario shown in Figure 1. Assume that, the recovery criterion associated with recovery point R1 specifies that resources (*numActiveUsersLock*) acquired after the recovery point should be released and does not care about occurrences of DR, LU or UR events. As shown in the Figure 5, the *getResourceOwnership* primitive is used to

determine ownership of the *numActiveUsersLock* resource. Then, if the restoration level indicates that a DR or LU event has occurred, that would imply that the thread has successfully completed incrementing *numActiveUsers* in the first place. Then in order to rollback the failed thread execution correctly to recovery point R1 without losing the work done by other threads, a matching decrement operation would need to be performed. If however the change track logs indicate that no other thread has consumed data written by the failed thread, it could imply that the failed thread either did not complete its increment operation or was the last thread to update the value of *numActiveUsers*. In that case, the recoverable thread could use its undo log to undo its changes, if any. The developer of this recovery handler is expected to have used the Undo log interfaces to store the old value prior to modification. Once state restoration is complete, execution is transferred to recovery point R1.

Similarly, in the case of the example in Figure 2, assume that the recovery criterion only specifies the constraint on releasing the temporary resource acquired after the recovery point. Therefore, the *getResourceOwnership* primitive is used to obtain the current ownership status of the temporary resource. If the resource is held by the thread, in order to rollback to recovery point R3, the resource must be cleanly relinquished. The pseudo code for this example and the next is not shown due to lack of space.

In the case of the failure scenario shown in Figure 3, the recovery criterion for recovery point R4 would be that no resources acquired after the recovery point (such as lock *MetadataLocationLock*) should be held by the thread and that no DR or LU events should have occurred. If the restoration level indicates that no other thread has already consumed this value (i.e., no DR or LU events have occurred), then the changes of the failed thread can be undone safely by replacing with the values in the Undo log. However, if the value is likely to have been consumed by another thread (i.e. DR or LU occurred), then the restoration level does not meet the recovery criterion for R4. So, in accordance with SRP, the error cannot be handled using single-thread recovery. Depending upon the support for multi-thread recovery (provided the CD event has not occurred) recovery may require rollbacks of multiple threads. If however, CD has occurred, then system-level recovery or error-compensation is performed.

4.3 Implementation Details

Undo logs go out of scope i.e., can be purged when a recoverable thread completes execution. Similarly, change track logs for a lock are purged when the recoverable thread issues a *stopTracking* call. How-

ever, unlike undo logs, change track logs cannot be purged immediately since these centralized logs may be shared by multiple recoverable threads. In that case, the log entries corresponding to the purging thread are only marked for purging and are actually purged when the last recoverable thread using the log issues a *stopTracking* call on that lock.

Multi-thread recovery i.e., applying state restoration and recovery to more than one thread, can typically handle more failure scenarios compared to single-thread recovery. However, multi-thread recovery is complex to implement. Moreover, multi-thread recovery may result in a domino effect [22] (also referred to as cascading aborts) potentially resulting in unavailability of resources and unbounded recovery time[6]. A simpler and more effective technique would be to limit recovery to a single thread and ensure recovery success through other mechanisms such as dependency tracking and scheduling. Recovery conscious scheduling [6] describes an approach where dependencies between concurrent threads are identified and dependent threads serialized. This approach can help limit the number of concurrent dependent threads and increase single-thread recovery success.

5 Experiments

We have implemented the Log(Lock) architecture for system state restoration and micro-recovery on an industry standard, high-performance storage controller and applied Log(Lock) to a variety of state and resource locks. In this section, we present our evaluation of Log(Lock) with respect to performance, failure recovery and scalability. We next describe our experimental setup, evaluation metrics, experimentation methodology and results.

We identified state and resource instances that are changed or accessed rapidly through the observation periods, based on instrumenting the system (Table 2). We also identified representative failure scenarios by analyzing bug reports, failure logs and code. Using these scenarios as candidates for micro-recovery and state restoration, we evaluate Log(Lock) efficiency and effectiveness. In summary, our results show that:

- The Log(Lock) architecture imposes negligible overhead and sustains high performance (< 10% impact) under a variety of workloads, even while tracking rapidly changing state (nearly 15K times/second) for significant durations.
- We observe an extremely high rate of recovery success (>99%), i.e., percentage of time restoration levels meet recovery criterion. This high rate of recovery success makes it evident that micro-recovery with Log(Lock) can be a promising ap-

proach to system recovery from transient failures.

- The Log(Lock) approach exhibits significant improvement in availability, replacing a four second downtime without micro-recovery with only a 35% performance impact lasting six seconds with Log(Lock).

5.1 Experimental Setup

We implemented the Log(Lock)-based state restoration architecture in an enterprise-class high performance, highly concurrent embedded storage controller. The system consists of a 4-way processor complex (4 3.00 GHz Xeon 5160 processors with 12 GB memory running IBM MCP Linux) running the controller software over a simulated backend. The controller implements persistent memory (non-volatile storage) for write caching. Simulating the backend allows flexibility in terms of experimenting with different configurations such as read/write latencies and error injection. The back end configuration varied between 50-250 disks of 100GB each with the maximum read and write latencies of the disk set to 20 ms. The memory footprint of our implementation of the Log(Lock) architecture was less than 48KB. The host functionality was performed from a different system (2 1.133 GHz Pentium III processor with 1 GB memory, RHLinux 9) connected to the storage complex through a high-bandwidth (2 GB) fiber channel interconnect.

Our workload was generated using a randomized synthetic workload generator which took the following inputs: read/write ratio, block size and queue depth (i.e. maximum number of outstanding requests from the host). The experiments presented in this paper utilized three distinct read/write ratios: 100% writes, 50%-50% mix of reads and writes and 100% reads. Block size was set to 4 KB and queue depth varied between 16 and 256.

5.2 Metrics

Our experiments evaluate efficiency and effectiveness of the Log(Lock) architecture. Efficiency and effectiveness depend on the following parameters: (1) rate of access to shared state or resources and (2) duration of a recoverable thread. Increasing each of these parameters results in an increase in the log size, logging overhead and the probability of conflicts.

Efficiency refers to the impact of Log(Lock) on system performance. To measure performance, we utilize two metrics: *throughput* (IOs per second or IOps) and *latency* (seconds/IO).

Effectiveness refers to the ability of the state restoration architecture to reduce the recovery time and positively impact the availability of the system.

Concretely, it refers to the probability of recovery success with the Log(Lock) architecture and the impact on system recovery time.

Effectiveness is measured using the following metrics: (1) *recovery success*, i.e. the percentage of time the restoration level meets the recovery criterion for single thread recovery, and (2) *recovery time*, i.e. the time required to restore the system to a consistent state after encountering a failure. Note that in the experiments reported in this paper we focus on single thread recovery while evaluating recovery success. While our Log(Lock) approach can also be applied to multi-thread recovery, as described in Section 4.3, multi-thread recovery can be costly in terms of coding effort, resource consumption and recovery time. Instead, we assume that a technique such as recovery conscious scheduling [6] can help reduce the need for multi-thread recovery and improve the success of single thread recovery.

5.3 Methodology

In order to evaluate Log(Lock), we first identify state and resource instances in the software for tracking. We instrumented the system to identify top locks in terms of access and contention. Table 2 shows the top five locks in terms of number of accesses and contention. The table shows the semantics of the lock (i.e. the state or resource protected), the number of CPU cycles lost to contention, number of occurrences of contention (> 2000 CPU cycles), number of accesses to the lock and the average number of lock acquisitions per IO. Frequently acquired locks are indicative of state that is accessed or modified often. For example, Table 2 shows that the fiber channel lock accessed nearly 10 times per IO is a good candidate for evaluating the efficiency of Log(Lock). Contention, while indicative of longer durations of holding locks, also shows a higher probability of accesses by concurrent threads. As Table 2 shows, the percentage of accesses resulting in lock contention is low as a result of the highly concurrent design of the controller. Thus, for short durations of tracking we expect high recovery success.

To evaluate effectiveness, we first measure the recovery success for the candidates identified from Table 2. We measure recovery success across locks with different rates of access and varying duration of tracking. To evaluate the impact on recovery time, we identify candidates for state restoration based on analysis of the software, failure logs and defects.

We present evaluation of the efficiency of our Log(Lock) architecture as compared to the original system, henceforth referred to as *baseline*. The baseline implementation does not perform state restoration or fine-grained recovery. Instead, it uses a highly efficient system level recovery mechanism that

Table 2: Lock Access over 75 minutes

Lock	Contention Cycles (Count)	Number of accesses	Locks/IO
Fiber channel	2654991 (578)	137196747	10.34
IO state	219969 (76)	90122610	6.79
Resource	608103 (100)	63482290	4.78
Resource state	124965 (52)	30040757	2.26
Throttle timer	79848 (11)	113316	0.0085

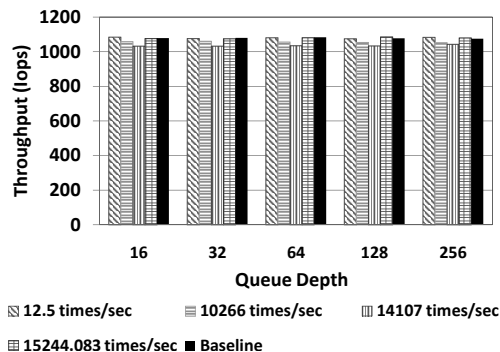


Figure 6: Rate vs Throughput (100% Writes)

checks all persistent system structures such as non-volatile data in the write cache for consistency, reinitializes software state and redrives lost tasks. Note that no hardware reboot is involved.

An alternative approach to Log(Lock) is to implement schemes such as strict 2-phase locking (2PL), commonly used in transactional systems. Essentially, these protocols require locks to be held for the entire duration of a recoverable thread. However, due to the high degree of concurrency in the system and the implementation of lock timeouts, such a scheme when implemented in our storage controller software caused lock timeouts and failed to bring up the system. Therefore, throughout this evaluation section, we primarily use the baseline system for comparison.

5.4 Efficiency of Log(Lock)

In order to measure efficiency, we compare the performance of the Log(Lock) architecture with the baseline system during failure-free operation.

5.4.1 Effect of Frequency of State Change

As described in Section 5.2, as the rate of accesses to a state variable or resource being tracked increases, the logging overhead increases. The workloads used for this experiment consisted of 100% write IOs and the data is averaged over 10 runs of 10 minutes each. The queue depth is represented on the x-axis. For this experiment, we chose four locks from Table 2, representative of a range of access rates, ranging from 12.5 times/second to 15244 times/second. The

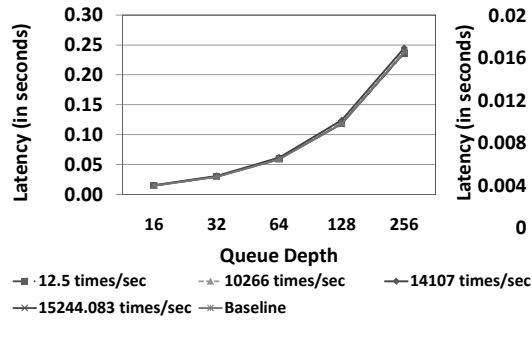


Figure 7: Rate vs Latency (100% Writes)

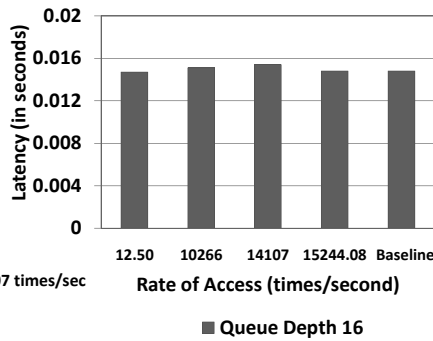


Figure 8: Latency

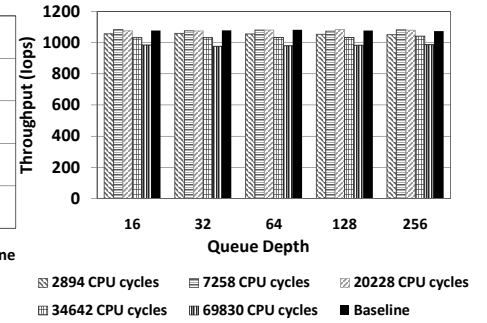


Figure 9: Duration of Tracking vs Throughput (100% Writes)

Table 3: % Duration of Tracking vs Latency

Queue Depth	(Duration of tracking in CPU Cycles)				
	2894	7258	20228	34642	69830
	% Increase in latency over baseline				
16	2.03%	0.68%	0.00%	4.05%	9.46%
32	1.69%	0.34%	0.34%	4.39%	10.47%
64	2.72%	0.34%	0.51%	4.76%	10.71%
128	2.54%	0.85%	0.00%	5.08%	9.32%
256	2.10%	0.00%	0.42%	2.94%	8.82%

duration of tracking was 2600 CPU cycles on average (and standard deviation 265 CPU cycles).

Figure 6 shows the throughput with varying access rates under different queue depths. The numbers show that even for high access rates, the Log(Lock) approach has negligible impact on performance. The lock with access rate 14107 times/sec (the resource pool lock) was tracked for 2429 CPU cycles and results in a 4.5% drop in throughput. We attribute this to the occurrence of nested lock conditions in that particular code path, causing the system to be sensitive to even the small delay introduced by Log(Lock).

Figure 7 shows the variation of latency with queue depth for different access rates. The curves for the various access rates almost completely overlap showing that across configurations, the impact of Log(Lock) on latency, even for high access rates, is negligible. The observation that the latency increases with queue depth is a queuing effect commonly observed in systems [23] and is independent of Log(Lock). Figure 8 zooms into the points for queue depth 16 to give the reader a closer look at the data. As in the case of throughput, latency increases by $\sim 4\%$ for the resource pool lock and is attributed to the occurrence of nested lock situations in the code path. The important message from Figures 6 and 7 is that Log(Lock) tracking can sustain high performance even while tracking rapidly modified/accessed state or resources.

Table 4: % Overhead (other workloads)

Queue Depth	Workload 1		Workload 2	
	Through-put	Latency	Through-put	Latency
16	0.43%	0.47%	0.08%	$\sim 0.00\%$
32	0.25%	$\sim 0.00\%$	0.78%	0.75%
64	0.24%	0.39%	0.13%	$\sim 0.00\%$
128	0.29%	0.39%	0.79%	0.75%
256	0.25%	0.00%	0.12%	0.19%

5.4.2 Effect of Duration of Tracking

Figures 9 and Table 3 show the variation of system performance with different durations of tracking. The durations were measured in terms of number of CPU cycles between the *startTracking* and *stopTracking* calls, averaged over 10 runs of 10 minutes each. The independent parameter queue depth is shown on the x-axis. The data represents the performance for candidate locks from Table 2 that were tracked for different durations ranging from 2894 CPU cycles to 69830 CPU cycles (IO state for 2894 and 69830 CPU cycles, timer, fiber channel and resource pool for 7258, 20228 and 34642 CPU cycles respectively). The numbers were chosen to be representative of a range of tracking durations. Since no functional code was modified, rather than varying the duration of a single lock, different locks were instrumented to obtain this range. The rate of access of each lock varied as shown in Table 5.

From Figures 9 and Table 3 we observe that, the performance of the system with Log(Lock) is comparable to the baseline system across various queue depths. For the IO state lock (a lock in the IO path), when the duration of tracking was increased from 2894 CPU cycles to 69830 CPU cycles, the throughput dropped by 8.85% and response time increased by 9.76% on average compared to baseline. This drop in performance can be attributed to two factors: (1) occurrence of more conflicts with increase in duration of tracking and (2) increased possibility

of encountering nested lock conditions, which are sensitive to the delay introduced by tracking. In the case of the resource lock, a tracking duration to 34642 CPU cycles resulted in a drop of only 4.3%, which is nearly identical to the performance with a tracking duration of only 2429 CPU cycles, as shown in Section 5.4.1. We conclude that, though the overhead of tracking is a function of both the frequency and duration of tracking, it is more significantly impacted by the semantics of the lock being tracked and the efficiency of the code path involving the lock.

5.4.3 Performance with Other Workloads

Table 4 show the throughput and latency with four other workloads. The figures compare the performance of a system powered by Log(Lock) and the baseline system under varying queue depths for the following workloads: Workload-1 (100% read, disk latency 1ms), and Workload-2 (50% read, disk latency 1ms). Data from tracking the fiber channel lock (15244 times/sec for 20228 CPU cycles each) is shown. Overall, the impact on performance was < 1% in all cases. These results reiterate the observation that Log(Lock) is lightweight and sustains high performance for a range of workloads.

Examining the object code for our implementation showed that in the event of a lock being tracked, fewer than 200 assembly instructions were added to the code path. Assuming one instruction executes per CPU cycle, even at a frequency of 15244 times/second, on a 3.00 GHz processor, this amounts to a time overhead of less than 1% (assuming that the size of the state being saved to undo logs is small). Also, note that storage controller code by itself is aggressively optimized to sustain high throughput, minimize the duration of locks in the I/O path and avoid nesting of locks to a large extent. Unlike checkpoints, which require a large amount of state to be copied to stable storage, our techniques copy small amounts of relevant state and information in memory only. The combination of all these factors results in the Log(Lock) system being able to sustain high performance despite an extremely high frequency of access to shared state and resources. In conclusion, we believe that the scenarios where performance will be impacted by tracking are when there are multiple levels of nesting with frequently accessed locks, increasing sensitivity to tracking delay. However, we expect that these situations are uncommon in well-designed systems.

5.5 Effectiveness of Log(Lock)

The next set of experiments are focused on evaluating the effectiveness of a micro-recovery framework with Log(Lock) in improving system recovery.

5.5.1 Recovery Success

The first metric of effectiveness is recovery success i.e., the percentage of time the restoration level meets the recovery criterion at the end of execution of a recoverable thread. This metric demonstrates the opportunity for micro-recovery in the system and evaluates if the system can effectively utilize Log(Lock)-based state restoration. Table 5 shows the recovery success for locks of varying semantics, rates of access and duration of tracking. The IO state lock was tracked for two types of recoverable threads, for a duration of 2894 CPU cycles in one and 69830 CPU cycles in the other. Hence data for this lock appears twice in Table 5. For each lock, the recovery criterion, the number of tracking threads per second, the rate of access, duration of tracking and recovery success are shown. The restoration level in each case was obtained by calling the *getRestorationLevel* method before *stopTracking*, and recovery success was computed as the percentage of time the restoration level met the recovery criterion. As Table 5 shows, our storage controller exhibits a high rate of recovery success for a range of locks, even with high rates of access. We conclude that, for failures involving the restoration of these instances of state and resources, fine-grained recovery presents an effective recovery strategy.

5.5.2 Recovery Time

To illustrate the impact of Log(Lock)-based micro-recovery on the overall recovery time and availability of the controller software, we injected transient failures that disappeared on retry. The failures required restoration of the IO state to its previous value and a retry of the function. For the Log(Lock) system, the recovery criterion for IO state was set as shown in Table 5. Once the failure was injected, the thread verified if the restoration level at the time of recovery met the recovery criterion, before attempting state restoration and retry. The tracking duration was equivalent to the set up with 69830 CPU cycles.

Figures 10 and 11 show the variation of throughput and latency respectively over time. The points of failure injection are marked in the figures. The throughput and latency shown are for a workload with 100% write IOs, queue depth 64 and disk latency 20 ms. The Log(Lock) architecture is compared to system-level recovery (abbreviated as SLR) in the case of the baseline system. Recall that SLR is implemented entirely in software and involves restarting the controller process and verifying data structures and cache data for consistency before redriving IO transactions. Overall, during failure-free operation, the average throughput and latency respectively with Log(Lock) is 708IOps,

Table 5: Recovery Success with the 100% Write Workload

Lock	Recovery Criterion	Tracking Calls (times/sec)	#Access (times/sec)	Duration CPU cycles	Recovery Success
Fiber channel	No Residual Resources	3666	15244	20228	100%
IO state	No DR, LU or UR	2500	10266	2894	99.88%
Resource pool	No Residual Resources	10	14107	34642	100%
Resource state	No Residual Resources	5	6675	4806	100%
Throttle timer	No Residual Resources	10	12.59	7258	100%
IO state	No DR, LU or UR	2444	10045	69830	99.38%

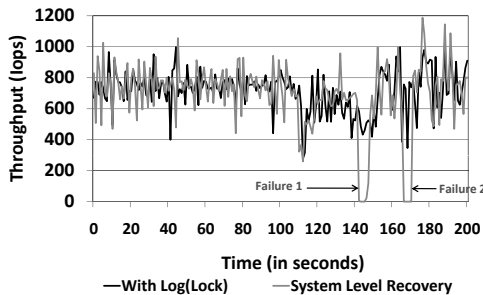


Figure 10: Throughput with Error Injection

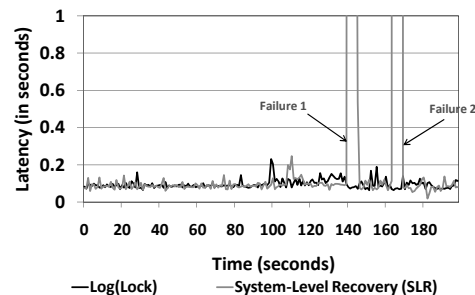


Figure 11: Latency with Error Injection

0.0946 sec/IO and 710IOPS, 0.0912 sec/IO for the baseline system.

Log(Lock)-enabled micro-recovery imposes a 35% performance overhead lasting six seconds during recovery. However, system-level recovery results in 4 seconds downtime and it takes an additional 2 seconds to begin sustaining high performance. It is important to remember that as the size of the system and in-memory data structures increase, the recovery time for SLR is bound to increase. This, along with the opportunity for micro-recovery illustrated by the high recovery success shown in the previous experiment, further promote the case for micro-recovery in high performance systems like the storage controller.

6 Related Work

Our work is largely inspired by previous work in the area of transactional systems, software fault tolerance and system availability. Hardware redundancy and software redundancy [24], rejuvenation [9] or fault isolation approaches such as isolating VMs from the failure of other VMs [14] are complementary to our techniques and are already deployed in our setups. Since these approaches are targeted at handling failures at a different level they focus on a coarser granularity of recovery compared to our techniques. Failure-oblivious computing [25] introduces a novel method to handle failures - by ignoring them and returning possibly arbitrary values. This technique may be applicable to systems like search engines where a few missing results may go unnoticed, but is not an option in storage controllers where ig-

norning failures or returning arbitrary values could lead to data corruption.

Application-specific recovery mechanisms such as recovery blocks [22], and exception handling [26] are used in many software systems. Constructs such as try/throw/catch [27] can be used to transfer control to an exception handler and a similar exception model is used by our implementation. However such exception handling constructs alone are insufficient for performing micro-recovery which requires richer failure context information. The goal of the Log(Lock) architecture is to provide this context information and provide the developer with a set of guidelines to decide the precise way in which the system should be restored given the failure context.

Logging of access patterns has been used for deterministic replay [15, 16, 17] during debugging. However, in micro-recovery, there is no requirement to perform deterministic replay. Also, the purpose of logging access patterns in Log(Lock) is to identify recovery dependencies between concurrent threads.

7 Conclusion

We have presented Log(Lock), a practical and flexible architecture for tracking dynamic dependencies and performing state restoration without rearchitecting legacy code. By exploring system state space, we formally model thread dependencies based on both state and shared resources, capturing failure contexts through different ‘restoration levels’. We develop recovery strategies in the form of restoration protocols based on recovery points and restoration levels. A comprehensive experimental evalua-

tion shows that Log(Lock)-enabled micro-recovery is both efficient and effective in reducing system recovery time.

Even with retrofittable mechanisms such as micro-recovery, we emphasize that failure recovery should be a design concern. One approach to reducing recovery time would be to design the software using components with independent failure modes (e.g. client-server interactions) or use a state space based approach where transitions to functional states can be identified even from a failure state.

Our effort in designing scalable failure recovery continues along a number of directions. One of our ongoing efforts is to reduce the need for programmer intervention in defining recovery actions. We are also interested in deploying and evaluating Log(Lock) in other high performance systems.

References

- [1] D. Scott, "Assessing the costs of application downtime.," *Gartner Group, Stamford, CT*, 1998.
- [2] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID hierarchical storage system," in *SOSP*, 1995.
- [3] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *SIGMOD Rec.*, vol. 17, no. 3, 1988.
- [4] "Cdp buyers guide," *Available at http://www.snia.org/tech_activities/dmf/docs/CDP_Buyers_Guide_20050822.pdf*, 2005.
- [5] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "IRON file systems," *SOSP*, 2005.
- [6] S. Seshadri, L. Chiu, C. Constantinescu, S. Balachandran, C. Dickey, L. Liu, and P. Muench, "Enhancing storage system availability on multi-core architectures using recovery conscious scheduling," in *USENIX FAST*, 2008.
- [7] D. Scott, "Making smart investments to reduce unplanned downtime.," *Tactical Guidelines Research Note, Gartner Group, Stamford, CT*, 1999.
- [8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot—a technique for cheap recovery," *OSDI*, 2004.
- [9] N. Kolettis and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *FTCS*, 1995.
- [10] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.
- [11] J. Gray and A. Reuter, *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann, October 1992.
- [12] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating bugs as allergies — a safe method to survive software failure," in *SOSP*, Oct 2005.
- [13] E. N. Elnozahy and J. S. Plank, "Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 2, pp. 97–108, 2004.
- [14] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems," in *USENIX ATC*, 2007.
- [15] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: a lightweight extension for rollback and deterministic replay for software debugging," in *USENIX ATC*, 2004.
- [16] M. Ronsse and K. D. Bosschere, "Recplay: a fully integrated practical record/replay system," *ACM TOCS*, vol. 17, no. 2, pp. 133–152, 1999.
- [17] M. Russinovich and B. Cogswell, "Replay for concurrent non-deterministic shared-memory applications," in *PLDI*, 1996.
- [18] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM TODS*, vol. 17, no. 1, pp. 94–162, 1992.
- [19] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.
- [20] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, 1978.
- [21] L. L. Pullum, *Software fault tolerance techniques and implementation*. Norwood, MA, USA: Artech House, Inc., 2001.
- [22] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software*, 1975.
- [23] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Prentice Hall PTR, 1982.
- [24] J. Gray, "Why do computers stop and what can be done about it?," in *Symposium on Reliability in Distributed Software and Database Systems*, 1986.
- [25] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebe, "Enhancing server availability and security through failure-oblivious computing," in *OSDI*, 2004.
- [26] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh, "Using rescue points to navigate software recovery," in *SP*, 2007.
- [27] B. Stroustrup, *The Design and Evolution of C++*. Addison-Wesley, 1994.

CLIC: CLient-Informed Caching for Storage Servers

Xin Liu
University of Waterloo

Ashraf Aboulnaga
University of Waterloo

Kenneth Salem
University of Waterloo

Xuhui Li
University of Waterloo

Abstract

Traditional caching policies are known to perform poorly for storage server caches. One promising approach to solving this problem is to use hints from the storage clients to manage the storage server cache. Previous hinting approaches are ad hoc, in that a predefined reaction to specific types of hints is hard-coded into the caching policy. With ad hoc approaches, it is difficult to ensure that the best hints are being used, and it is difficult to accommodate multiple types of hints and multiple client applications. In this paper, we propose CLient-Informed Caching (CLIC), a generic hint-based policy for managing storage server caches. CLIC automatically interprets hints generated by storage clients and translates them into a server caching policy. It does this without explicit knowledge of the application-specific hint semantics. We demonstrate using trace-based simulation of database workloads that CLIC outperforms hint-oblivious and state-of-the-art hint-aware caching policies. We also demonstrate that the space required to track and interpret hints is small.

1 Introduction

Multi-tier block caches arise in many situations. For example, running a database management system (DBMS) on top of a storage server results in at least two caches, one in the DBMS and one in the storage system. The challenges of making effective use of caches below the first tier are well known [15, 19, 22]. Poor temporal locality in the request streams experienced by the second-tier caches reduces the effectiveness of recency-based replacement policies [22], and failure to maintain exclusivity among the contents of the caches in each tier leads to wasted cache space [19].

Many techniques have been proposed for improving the performance of second-tier caches. Section 7 provides a brief survey. One promising class of techniques relies on *hinting*: the application that manages the first-

tier cache generates hints and attaches them to the I/O requests that it directs to the second tier. The cache at the second tier then attempts to exploit these hints to improve its performance. For example, an *importance hint* [6] indicates the priority of a particular page to the buffer cache manager in the first-tier application. Given such hints, the second-tier cache can infer that pages that have high priority in the first tier are likely to be retained there, and can thus give them low priority in the second tier. As another example, a *write hint* [11] indicates whether the first tier is writing a page to ensure recoverability of the page, or to facilitate replacement of the page in the first-tier cache. The second tier may infer that replacement writes are better caching candidates than recovery writes, since they indicate pages that are eviction candidates in the first tier.

Hinting is valuable because it is a way of making application-specific information available to the second (or lower) tier, which needs a good basis on which to make its caching decisions. However, previous work has taken an *ad hoc* approach to hinting. The general approach is to identify a specific type of hint that can be generated from an application (e.g., a DBMS) in the first tier. A replacement policy that knows how to take advantage of this particular type of hint is then designed for the second tier cache. For example, the TQ algorithm [11] is designed specifically to exploit write hints. The desired response to each possible hint is hard-coded into such an algorithm.

Ad hoc algorithms can significantly improve the performance of the second-tier cache when the necessary type of hint is available. However ad hoc algorithms also have some significant drawbacks. First, because the response to hints is hard-coded into an algorithm at the second tier, any change to the hints requires changes to the cache management policy at the second-tier server. Second, even if change is possible at the server, it is difficult to generalize ad hoc algorithms to account for new situations. For example, suppose that applications can gen-

erate *both* write hints and importance hints. Clearly, a low-priority (to the first tier) replacement write is probably a good caching candidate for the second tier, but what about a low-priority recovery write? In this case, the importance hint suggests that the page is a good candidate for caching in the second tier, but the write hint suggests that it is a poor candidate. One response to this might be to hard code into the second-tier cache manager an appropriate behavior for all combinations of hints that might occur. However, each new type of hint will multiply the number of possible hint combinations, and it may be difficult for the policy designer to determine an appropriate response for each one. A related problem arises when multiple first-tier applications are served by a single cache in the second tier. If different applications generate hints, how is the second tier cache to compare them? Is a write hint from one application more or less significant than an importance hint from another?

In this paper, we propose CLient-Informed Caching (CLIC), a *generic technique* for exploiting application hints to manage a second-tier cache, such as a storage server cache. Unlike ad hoc techniques, CLIC does not hard-code responses to any particular type of hint. Instead, it is an adaptive approach that attempts to learn to exploit any type of hint that is supplied to it. Applications in the first tier are free to supply any hints that they believe may be of value to the second tier. CLIC analyzes the available hints and determines which can be exploited to improve second-tier cache performance. Conversely, it learns to ignore hints that do not help. Unlike ad hoc approaches, CLIC decouples the task of generating hints (done by applications in the first tier) from the task of interpreting and exploiting them. CLIC naturally accommodates applications that generate more than one type of hint, as well as scenarios in which multiple applications share a second-tier cache.

The contributions of this paper are as follows. First, we define an on-line cost/benefit analysis of I/O request hints that can be used to determine which hints provide potentially valuable information to the second-tier cache. Second, we define an adaptive, priority-based cache replacement policy for the second-tier cache. This policy exploits the results of the hint analysis to improve the hit ratio of the second-tier cache. Third, we use trace-based simulation to provide a performance analysis of CLIC. Our results show that CLIC outperforms ad hoc hinting techniques and that its adaptivity can be achieved with low overhead.

2 Generic Framework for Hints

We assume a system in which multiple storage server client applications generate requests to a storage server, as shown in Figure 1. We are particularly interested in

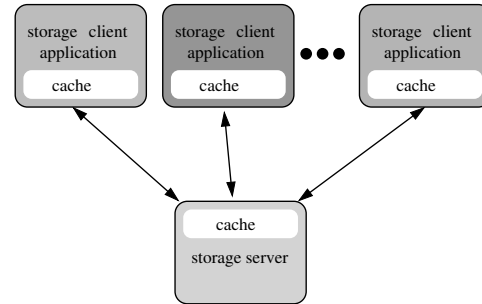


Figure 1: System Architecture

client applications that cache data, since it is such applications that give rise to multi-tier caching.

The storage server's workload is a sequence of block I/O requests from the various clients. When a client sends an I/O request (read or write) to the server, it may attach hints to the request. Specifically, each storage client may define one or more *hint types* and, for each such hint type, a *hint value domain*. When the client issues an I/O request, it attaches a *hint set* to the request. Each hint set consists of one hint value from the domain of each of the hint types defined by that client. For example, we used IBM DB2 Universal Database¹ as a storage client application, and we instrumented DB2 so that it would generate five types of hints, as described in the first five rows of Figure 2. Thus, each I/O request issued by DB2 will have an attached hint set consisting of 5 hint values: a pool ID, an object ID, an object type ID, a request type, and a DB2 buffer priority.

CLIC does *not* require these specific hint types. We chose these particular types of hints because they could be generated easily from DB2, and because we believed that they might prove useful to the underlying storage system. Each application can generate its own types of hints. CLIC itself only assumes that the hint value domains are *categorical*. It neither assumes nor exploits any ordering on the values in a hint value domain. Each storage client application may have its own hint types. In fact, even if two storage clients are instances of the same application (e.g., two instances of DB2) and use the same hint types, CLIC treats each client's hint types as distinct from the hint types of all other clients.

3 Hint Analysis

Every I/O request, read or write, represents a caching opportunity for the storage server. The storage server must decide whether to take advantage of each such opportunity by caching the requested page. Our approach is to base these caching decisions on the hint sets supplied by the client applications with each I/O request. CLIC associates each possible hint set H with a numeric priority,

DBMS	Hint Type	Value Domain Cardinality (TPC-C)	Value Domain Cardinality (TPC-H)	Description
DB2	pool ID	2	5	Identifies which DB2 buffer pool generated the I/O request.
DB2	object ID	21	23	Identifies a group of related database objects, such as a table and its associated indices.
DB2	object type ID	6	9	Identifies object type, such as table or index. Together, a pool ID, object ID and object type ID uniquely identify a database object.
DB2	request type	5	5	For read requests, distinguishes regular reads from prefetch reads. For writes, provides write hints ([11]), which distinguish between recovery writes, replacement writes, and synchronous writes. Synchronous writes are replacement writes that are not performed by an asynchronous page cleaning thread.
DB2	buffer priority	4	1	Identifies the priority of the page in its DB2 buffer cache.
MySQL	thread ID	-	5	ID of server thread that issued the request.
MySQL	request type	-	3	Read, replacement write, or recovery write.
MySQL	file ID	-	9	MySQL is configured so that each table is stored in a separate file, together with any indexes defined on that table, so this hint distinguishes groups of database objects.
MySQL	fix count	-	2	indicates how many MySQL threads are have currently fixed (pinned) this page in the buffer pool

Figure 2: Types of Hints in the DB2 and MySQL I/O Request Traces

$\text{Pr}(H)$. When an I/O request (read or write) for page p with attached hint set H arrives at the server, the server uses $\text{Pr}(H)$ to decide whether to cache p . Cache management at the server will be described in more detail in Section 4, but the essential idea is simple: the server caches p if there is some page p' in the cache that was requested with a hint set H' for which $\text{Pr}(H') < \text{Pr}(H)$.

We expect that some hint sets may signal pages that are likely to be re-used quickly, and thus are good caching candidates. Other hint sets may signal the opposite. Intuitively, we want the priority of each hint set to reflect these signals. But how should priorities be chosen for each hint set? One possibility is to assign these priorities, in advance, based on knowledge of the client application that generates the hint sets. Most existing hint-based caching techniques use this approach. For example, the TQ algorithm [11], which exploits write hints, understands that replacement writes likely indicate evictions in the client application's cache, and so it gives them high priority.

CLIC takes a different approach to this problem. Instead of predefining hint priorities based on knowledge of the storage client applications, CLIC assigns a priority to each hint set by *monitoring and analyzing I/O requests that arrive with that hint set*. Next, we describe

how CLIC performs its analysis.

We will assume that each request that arrives at the server is tagged (by the server) with a sequence number. Suppose that the server gets a request (p, H) , meaning a request (read or a write) for a page p with an attached hint set H , and suppose that this request is assigned sequence number s_1 . CLIC is interested in whether and when page p will be requested again after s_1 . There are three possibilities to consider:

write re-reference: The first possibility is that the *next* request for p in the request stream is a write request occurring with sequence number s_2 ($s_2 > s_1$). In this case, there would have been no benefit whatsoever to caching p at time s_1 . A cached copy of p would not help the server handle the subsequent write request any more efficiently. A cached copy of p may be of benefit for requests for p that occur after s_2 , but in that case the server would be better off caching p at s_2 rather than at s_1 . Thus, the server's caching opportunity at s_1 is best ignored.

read re-reference: The second possibility is that the *next* request for p in the request stream is read request at time s_2 . If the server caches p at time s_1 and keeps p in the cache until s_2 , it will benefit by

being able to serve the read request at s_2 from its cache. For the server to obtain this benefit, it must allow p to occupy one page “slot” in its cache during the interval $s_2 - s_1$.

no re-reference: The third possibility is that p is never requested again after s_1 . In this case, there is clearly no benefit to caching p at s_1 .

Of course, the server cannot determine which of these three possibilities will occur for any particular request, as that would require advance knowledge of the future request stream. Instead, we propose that the server base its caching decision for the request (p, H) on an analysis of previous requests with hint set H . Specifically, CLIC tracks three statistics for each hint set H :

$N(H)$: the total number of requests with hint set H .

$N_r(H)$: the total number requests with hint set H that result in a read re-reference (rather than a write re-reference or no re-reference).

$D(H)$: for those requests (p, H) that result in read re-references, the average number of requests that occur between the request and the read re-reference.

Using these three statistics, CLIC performs a simple benefit/cost analysis for each hint set H , and assigns higher priorities to hint sets with higher benefit/cost ratios. Suppose that the server receives a request (p, H) and that it elects to cache p . If a read re-reference subsequently occurs while p is cached, the server will have obtained a benefit from caching p . We arbitrarily assign a value of 1 to this benefit (the value we use does not affect the relative priorities of pages). Among all previous requests with hint set H , a fraction

$$f_{hit}(H) = N_r(H)/N(H) \quad (1)$$

eventually resulted in read re-references, and would have provided a benefit if cached. We call $f_{hit}(H)$ the *read hit rate* of hint set H . Since the value of a read re-reference is 1, $f_{hit}(H)$ can be interpreted as the expected benefit of caching and holding pages that are requested with hint set H . Conversely, $D(H)$ can be interpreted as the expected *cost* of caching such pages, as it measures how long such pages must occupy space in the cache before the benefit is obtained. We define the *caching priority* of hint set H as:

$$Pr(H) = \frac{f_{hit}(H)}{D(H)} \quad (2)$$

which is the ratio of the expected benefit to the expected cost.

Figure 3 illustrates the results of this analysis for a trace of I/O requests made by DB2 during a run of the

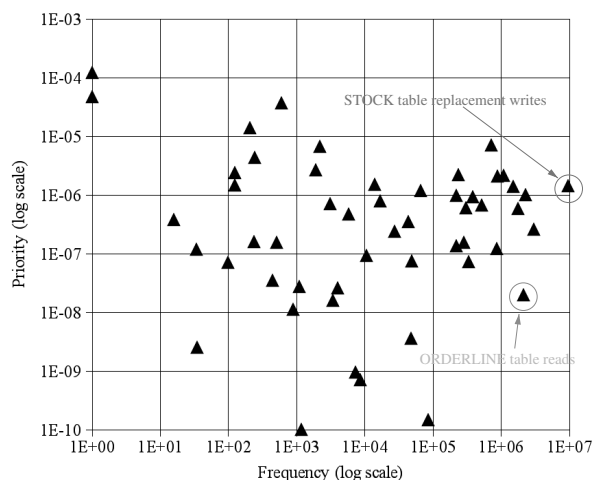


Figure 3: Hint Set Priorities for the DB2_C60 Trace
Each point represents a distinct hint set. All hint sets are shown.

TPC-C benchmark. Our workload traces will be described in more detail in Section 6. Each point in Figure 3 represents a distinct hint set that is present in the trace, and describes the hint set’s caching priority and frequency of occurrence. All hint sets with non-zero caching priority are shown. Clearly, some hint sets have much higher priorities, and thus much higher benefit/cost ratios, than others. For illustrative purposes, we have indicated partial interpretations of two of the hint sets in the figure. For example, the most frequently occurring hint set represents replacement writes to the STOCK table in the TPC-C database instance that was being managed by the DB2 client. We emphasize that CLIC does not need to understand that this hint represents the STOCK table, nor does it need to understand the difference between a replacement write and a recovery write. Its interpretation of hints is based entirely on the hint statistics that it tracks, and it can automatically determine that a request with the STOCK table hint set is a better caching opportunity than a request with the ORDERLINE table hint set.

3.1 Tracking Hint Set Statistics

To track hint set statistics, CLIC maintains a *hint table* with one entry for each distinct hint set H that has been observed by the storage server. The hint table entry for H records the current values of the statistics $N(H)$, $N_r(H)$ and $D(H)$. When the server receives a request (p, H) , it increments $N(H)$. Tracking $N_r(H)$ and $D(H)$ is somewhat more involved, as CLIC must determine whether a read request for page p is a read re-reference. To determine this, CLIC records two pieces of information for every page p that is cached: $seq(p)$, which is the sequence number of the most recent request for p , and

$H(p)$, which is the hint set that was attached to the most recent request for p . In addition, CLIC records $\text{seq}(p)$ and $H(p)$ for a fixed number (N_{outq}) of additional, uncached pages. This additional information is recorded in a data structure called the *outqueue*. N_{outq} is a CLIC parameter that can be used to bound the amount of space required for tracking read re-references. When the server receives a read request for page p with sequence number s , it checks both the cache and the outqueue for information about the most recent previous request, if any, for p . If it finds $\text{seq}(p)$ and $H(p)$ from a previous request, then it knows that the current request is a read re-reference of p . It increments $N_r(H(p))$ and it updates $D(H(p))$ using the re-reference distance $s - \text{seq}(p)$.

When a page p is evicted from the cache, an entry for p is inserted into the outqueue. An entry is also placed in the outqueue for any requested page that CLIC elects not to cache. (CLIC's caching policy is described in Section 4.) If the outqueue is full when a new entry is to be inserted, the least-recently inserted entry is evicted from the outqueue to make room for the new entry.

Since CLIC only records $\text{seq}(p)$ and $H(p)$ for a limited number of pages, it may fail to recognize that a new read request (p, H) is actually a read re-reference for p . Some error is inevitable unless CLIC were to record information about all requested pages. However, CLIC's approach to tracking page re-references has several advantages. First, since CLIC tracks the most recent reference to all pages that are in the cache, we expect to have accurate re-reference distance estimates for hint sets that are believed to have the highest priorities, since pages requested with those hint sets will be cached. If the priority of such hint sets drops, CLIC should be able to detect this. Second, by evicting the oldest entries from the outqueue when eviction is necessary, CLIC will tend to miss read re-references that have long re-reference distances. Conversely, read re-references that happen quickly are likely to be detected. These are exactly the type of re-references that lead to high caching priority. Thus, CLIC's statistics tracking is biased in favor of read re-references that are likely to lead to high caching priority.

3.2 Time-Varying Workloads

To accommodate time-varying workloads, CLIC divides the request stream into non-overlapping windows, with each window consisting of W requests. At the end of each window, CLIC adjusts the priority for each hint set using the statistics collected during that window. The adjusted priority will be used to guide the caching policy during the next window. It then clears the statistics ($N(H)$, $N_r(H)$, $D(H)$) for all hint sets in the hint table so that it can collect new statistics during the next window.

Let $\text{Pr}(H)_i$ represent the priority of H that is calculated after the i th window, and that is used by CLIC's caching policy during window $i + 1$. Priority $\text{Pr}(H)_i$ is calculated as follows

$$\text{Pr}(H)_i = r\widehat{\text{Pr}}(H)_i + (1 - r)\text{Pr}(H)_{i-1} \quad (3)$$

where $\widehat{\text{Pr}}(H)_i$ represents the priorities that were calculated using the statistics collected during the i th window (and Equation 2), and r ($0 < r \leq 1$) is a CLIC parameter. The effect of Equation 3 is that the impact of statistics gathered during the i th window decays exponentially with each new window, at a rate that is controlled by r . Setting $r = 1$ causes CLIC to base its priorities entirely on the statistics collected during the most recently completed window. Lower values of r cause CLIC to give more weight to older statistics. For all of the experiments reported in this paper, we have set $W = 10^6$ and $r = 1$.

4 Cache Management

In the previous section, we described how CLIC assigns a caching priority to each hint set H . In this section, we describe how the server uses these priorities to manage the contents of its cache.

Figure 4 describes CLIC's priority-based replacement policy. This policy evicts a lowest priority page from the cache if the newly requested page has higher priority. The priority of a page is determined by the priority $\text{Pr}(H)$ of the hint set H with which that page was last requested. Note that if a page that is cached after being requested with hint set H is subsequently requested with hint set H' , its priority changes from $\text{Pr}(H)$ to $\text{Pr}(H')$. The most recent request for each cached page always determines its caching priority.

The policy described in Figure 4 can be implemented to run in constant expected time. To do this, CLIC maintains a heap-based priority queue of the hint sets. For each hint set H in the heap, all pages with $H(p) = H$ are recorded in a doubly-linked list that is sorted by $\text{seq}(p)$. This allows the victim page to be identified (Figure 4, lines 7-11) in constant time. CLIC also maintains a hash table of all cached pages so that it can tell which pages are cached (line 1) and find a cached page in its hint set list in constant expected time. Finally, CLIC implements the hint table as a hash table so that it can look up $\text{Pr}(H)$ (line 12) in constant expected time.

As described in Section 3.2, CLIC adjusts hint set priorities after every window of W requests. When this occurs, CLIC rebuilds its hint set priority queue based on the newly adjusted priorities. Hint set priorities do not change except at window boundaries.

```

1  if p is not cached then
2    if the cache is not full then
3      cache p
4      set seq(p) = s
5      set H(p) = H
6    else
7      let m be the minimum priority
8        of all pages in the cache
9      let v be the page with the
10        minimum sequence number seq(v)
11        among all pages with priority m
12      if Pr(H) > m then
13        evict v from the cache
14        add entry for v (with seq(v)
15          and H(v)) to the outqueue
16        cache p
17        set seq(p) = s
18        set H(p) = H
19      else /* do not cache p */
20        add entry for p to the outqueue
21        set seq(p) = s
22        set H(p) = H
23    else /* p is already cached */
24      seq(p) = s
25      H(p) = H

```

Figure 4: Hint-Based Server Cache Replacement Policy
This pseudo-code shows how the server handles a request for page p with hint set H and request sequence number s .

5 Handling Large Numbers of Hint Sets

As described in Section 3.1, CLIC’s hint table records statistical information about every hint set that the server has observed. Although the amount of statistical information tracked per hint set is small, the number of distinct hit sets from each client might be as large as the product of the cardinalities of that client’s hint value domains. In our traces, the number of distinct hit sets is small. For other applications, however, the number of hint sets could potentially be much larger. In this section, we propose a simple technique for restricting the number of hint sets that CLIC must consider, so that CLIC can continue to operate efficiently as the number of hint sets grows.

All of the hint types in our workload traces exhibit frequency skew. That is, some values in the hint domain occur much more frequently than others. As a result, some hint sets occur much more frequently than others. To reduce the number of hints that CLIC must consider, we propose to exploit this skew by tracking statistics for the hint sets that occur most frequently in the request stream and ignoring those that do not. Ignoring infrequent hint sets may lead to errors. In particular, we may miss a hint set that would have had high caching priority. However, since any such missed hint set would occur infrequently,

the impact of the error on the server’s caching performance is likely to be small.

The problem with this approach is that we must determine, on the fly, which hint sets occur frequently, without actually maintaining a counter for every hint set. Fortunately, this *frequent item problem* arises in a variety of settings, and numerous methods have been proposed to solve it. We have chosen one of these methods: the so-called *Space-Saving* algorithm [14], which has recently been shown to outperform other frequent item algorithms [7]. Given a parameter k , this algorithm tracks the frequency of k different hint sets, among which it attempts to include as many of the actual k most frequent hint sets as possible. It is an on-line algorithm which scans the sequence of hint sets attached to the requests arriving at the server. Although k different hint sets are tracked at once, the specific hint sets that are being tracked may vary over time, depending on the request sequence.

After each request has been processed, the algorithm can report the k hint sets that it is currently tracking, as well as an estimate of the frequency (total number of occurrences) of each hint set and an error indicator which bounds the error in the frequency estimate. By analyzing the frequency estimates and error indicators, it is possible to determine which of the k currently-tracked hint sets are guaranteed to be among the actual top- k most frequent hint sets and which are not. However, for our purposes this is not necessary.

We adapted the Space-Saving algorithm slightly so that it tracks the additional information we require for our analysis. Specifically:

$N(H)$: For each hint set H that is tracked by the Space-Saving algorithm, we use the frequency estimate produced by the algorithm, minus the estimation error bound reported by the algorithm, as $N(H)$.

$N_r(H)$: We modified the Space-Saving algorithm to include an additional counter for each hint set H that is currently being tracked. This counter is initialized to zero when the algorithm starts tracking H , and it is incremented for each read re-reference involving H that occurs while H is being tracked. We use the value of this counter as $N_r(H)$.

$D(H)$: We track the expected re-reference distance for all read re-references involving H that occur while H is being tracked, i.e., those read re-references that are included in $N_r(H)$.

For all hint sets H that are not currently tracked by the algorithm, we take $N_r(H)$ to be zero, and hence $\Pr(H)$ to be zero as well.

In general, $N(H)$ will be an underestimate of the true frequency of hint set H . Since $N_r(H)$ is only incre-

mented while H is being tracked, it too will in general underestimate the true frequency of read re-references involving H . As a result of these underestimations, $f_{hit}(H)$, which is calculated as the ratio of the $N_r(H)$ to $N(H)$, may be inaccurate. However, because we take the ratio of $N(H)$ to $N_r(H)$, the two underestimations may at least partially cancel one another, leading to a more accurate $f_{hit}(H)$. In addition, the higher the true frequency of H , the more time H will spend being tracked and the more accurate we expect our estimates to be.

To account for time-varying workloads, we restart the Space-Saving algorithm from scratch for every window of W requests. Specifically, at the end of each window we use the Space-Saving algorithm to estimate $N(H)$, $N_r(H)$, and $D(H)$ for each hint set H that is tracked by the algorithm, as described above. These statistics are used to calculate $\widehat{Pr}(H)$, which is then used in Equation 3 to calculate the hint set's caching priority ($Pr(H)$) to be used during the next request window. Once the $\widehat{Pr}(H)$ have been calculated, the Space-Saving algorithm's state is cleared in preparation for the next window.

The Space-Saving algorithm requires two counters for each tracked hint-set, and we added several additional counters for the sake of our analysis. Overall, the space required is proportional to k . Thus, this parameter can be used to limit the amount of space required to track hint set statistics. With each new request, the data structure used by the Space-Saving algorithm can be updated in constant time [14], and the statistics for the tracked hint sets can be reported, if necessary, in time proportional to k .

6 Experimental Evaluation

Objectives: We used trace-driven simulation to evaluate our proposed mechanisms. The goal of our experimental evaluation is to answer the following questions:

1. Can CLIC identify good caching opportunities for storage server caches, and thereby improve the cache hit ratio in compared to other caching policies? (Section 6.1)
2. How effective are CLIC's mechanisms for reducing the number of hint sets that it must track (Sections 6.2 and 6.3).
3. Can CLIC improve performance for multiple storage clients by prioritizing the caching opportunities of the different clients based on their observed reference behavior? (Section 6.4)

Simulator: To answer these questions, we implemented a simulation of the storage server cache. In addition to CLIC, the simulator implements the following caching policies for purpose of comparison:

OPT: This is an implementation of the well-known optimal off-line MIN algorithm [4]. It replaces the cached page that will not be read for the longest time. This algorithm requires knowledge of the future so it cannot be used for cache replacement in practical systems, but its hit ratio is optimal so it serves as an upper bound on the performance of any caching algorithm.

LRU: This algorithm replaces the least-recently used page in the cache. Since temporal locality is often poor in second-tier caches, we expect CLIC to perform significantly better than LRU.

ARC: ARC [13] is a hint-oblivious caching policy that considers both recency and frequency of use in making replacement decisions.

TQ: TQ is a hint-aware algorithm that was proposed for use in second-tier caches [11]. Unlike the algorithms proposed here, it works only with one specific type of hint that can be associated with write requests from database systems. We expect our proposed algorithms, which can automatically exploit any type of hint, to do at least as well as TQ when the write hints needed by TQ are present in the request stream.

The TQ algorithm has previously been compared to a number of other second-tier caching policies that are not considered here. These include MQ [22], a hint-oblivious policy, and write-hint-aware variations of both MQ and LRU. TQ was shown to be generally superior to those alternatives when the necessary write hints are present [11], so we use it as our representative of the state of the art in hint-aware second-tier caching policies.

The simulator accepts a stream of I/O requests with associated hint sets, as would be generated by one or more storage clients. It simulates the caching behavior of one of the five supported cache replacement policies (CLIC, OPT, LRU, ARC and TQ) and computes the *read hit ratio* for the storage server cache. The read hit ratio is the number of read hits divided by the number of read requests.

Workloads: In this paper, we use DB2 Universal Database (version 8.2) and the MySQL² database system (Community Edition, version 5.0.33) as our storage system clients. DB2 is a widely-used commercial relational database system to which we had access to source code, and MySQL is a widely-used open source relational database system. We instrumented DB2 and MySQL so that they would generate I/O hints and dump them into an I/O trace. The types of hints generated by these two systems are described in Figure 2.

To generate our traces, we ran TPC-C and TPC-H workloads on DB2 and a TPC-H workload on MySQL.

Trace Name	DBMS	WkLoad	DB Size (pages)	DBMS Buffer Size (pages)	Requests	Distinct Hint Sets	Distinct Pages
DB2_C60	DB2	TPC-C	600K	60K	37699091	164	930688
DB2_C300	DB2	TPC-C	600K	300K	31869377	154	1320882
DB2_C540	DB2	TPC-C	600K	540K	21863719	140	1807431
DB2_H80	DB2	TPC-H	800K	80K	635375701	134	732905
HB2_H400	DB2	TPC-H	800K	400K	65675204	129	732723
DB2_H720	DB2	TPC-H	800K	720K	3077872	128	732690
MY_H65	MySQL	TPC-H	328K	65K	36266735	21	167502
MY_H98	MySQL	TPC-H	328K	98K	16561346	21	167501

Figure 5: I/O Request Traces. The page sizes for the DB2 and MySQL databases were 4KB and 16KB, respectively. For the TPC-C workloads, the table shows the initial database size. The TPC-C database grows as the workload runs.

TPC-C and TPC-H are well-known on-line transaction processing (TPC-C) and decision support (TPC-H) benchmarks. We ran TPC-C at scale factor 25. At this scale factor, the TPC-C database initially occupied approximately 600,000 4KB blocks, or about 2.3 GB, in the storage system. The TPC-C workload inserts new items into the database, so the database grows during the TPC-C run. For the TPC-H experiments, the database size was approximately 3.2 GB for the DB2 runs, and 5 GB for the MySQL runs. The DB2 TPC-H workload consisted of the 22 TPC-H queries and the two refresh updates. The workload for MySQL was similar except that it did not include the refresh updated and we skipped one of the 22 queries (Q18) because of excessive run-time on our MySQL configuration.

On each run, we controlled the size of the database system’s internal buffer cache. We collected traces using a variety of different buffer cache sizes for each DBMS. We expect the buffer cache size to be a significant parameter because it affects the temporal locality in the I/O request stream that is seen by the underlying storage server. Figure 5 summarizes the I/O request traces that were used for the experiments reported here.

6.1 Comparison to Other Caching Policies

In our first experiment, we compare the cache read hit ratio of CLIC to that of other replacement policies that we consider (LRU, ARC, TQ, and OPT). We varied the size of the storage server buffer cache, and we present the read hit ratio as a function of the server’s buffer cache size for each workload. For these experiments, we set $r = 1.0$ and the size of CLIC’s outqueue (N_{outq}) to 5 entries per page in the storage server’s cache. If the cache holds C pages, this means that CLIC tracks the most recent reference for $6C$ pages, since it tracks this information for all cached pages, plus those in the outqueue. For each tracked page, CLIC records a sequence number and a hint set. If each of these is stored as a 4-byte integer, this represents a space overhead of roughly 1%.

To account for this, we reduced the server cache size by 1% for CLIC only, so that the total space used by CLIC would be the same as that used by other policies. ARC also employs a structure similar to CLIC’s outqueue for tracking pages that are not in the cache. However, we did not reduce ARC’s cache size. As a result, ARC has a small space advantage in these experiments.

Figure 6 shows the results of this experiment for the DB2 TPC-C traces. All of the algorithms have similar performance for the DB2_C60 trace. That trace comes from the DB2 configuration with the smallest buffer cache, and there is a significant amount of temporal locality in the trace that was not “absorbed” by DB2’s cache. This temporal locality can be exploited by the storage server cache. As a result, even LRU performs reasonably well. Both of the hint-based algorithms (TQ and CLIC) also do well.

The performance of LRU is significantly worse on the other two TPC-C traces, as there is very little temporal locality. ARC performs better than LRU, as expected, though substantially worse than both of the hint-aware policies. CLIC, which learns how to exploit the available hints, does about as well as TQ, which implements a hard-coded response to one particular hint type on the DB2_C300 trace, and both policies’ performance approaches that of OPT. CLIC outperforms TQ on the DB2_C540 trace, though it is also further from OPT. The DB2_C540 trace comes from the DB2 configuration with the largest buffer cache, so it has the least temporal locality of all traces and therefore presents the most difficult cache replacement problem.

Figures 7 and 8 show the results for the TPC-H traces from DB2 and for the MySQL TPC-H traces, respectively. Again, CLIC generally performs at least as well as the other replacement policies that we considered. In some cases, e.g., for the DB2_H400 trace, CLIC’s read hit ratio is more than twice the hit ratio of the best hint-oblivious alternative. In one case, for the DB2_H80 trace with a server cache size of 300K pages, both LRU and

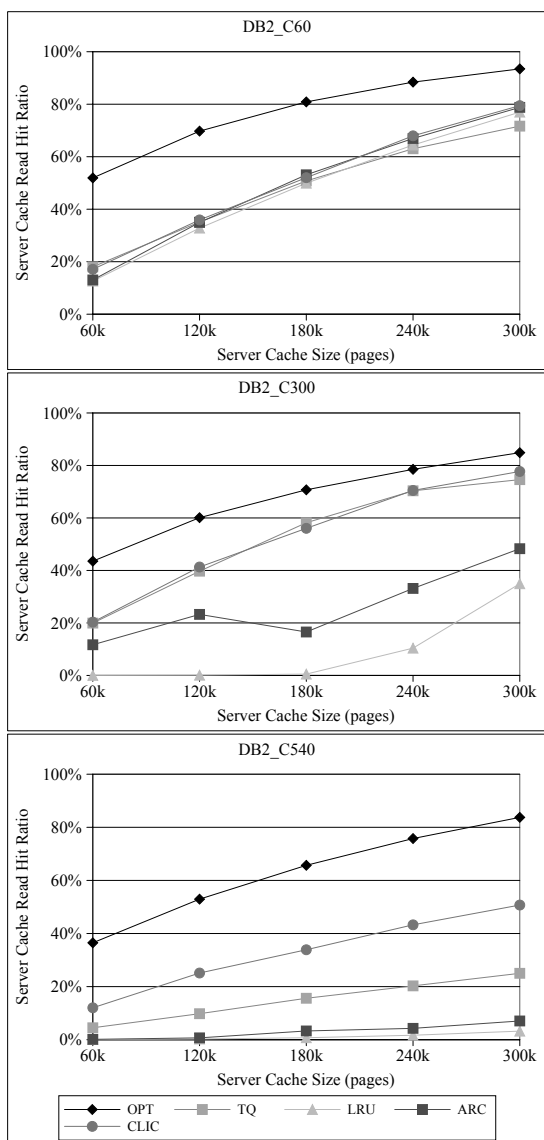


Figure 6: Read Hit Ratio of Caching Policies for the DB2 TPC-C Workloads

ARC outperformed both TQ and CLIC. We are uncertain of the precise reason for this inversion. However, this is a scenario in which there is a relatively large amount of residual locality in the workload (because the DB2 buffer cache is small) and in which the storage server cache may be large enough to capture it.

6.2 Tracking Only Frequent Hint Sets

In this experiment, we study the effect of tracking only the most frequently occurring hint sets using the top- k algorithm described in Section 5. In our experiment we vary k , the number of hint sets tracked by CLIC, and measure the server cache hit ratio.

Figure 9 shows some of the results of this experiment.

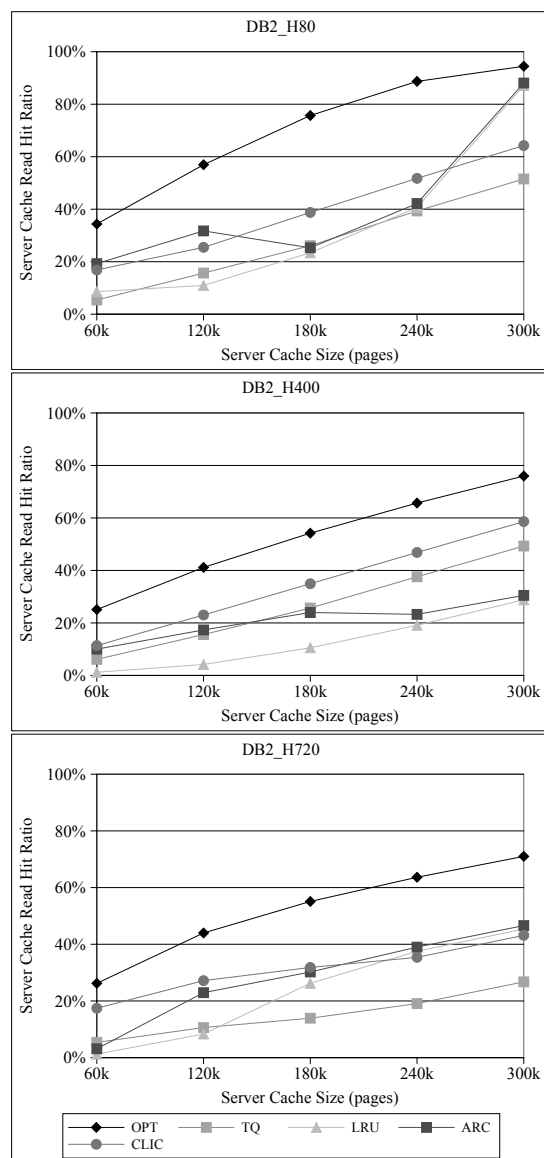


Figure 7: Read Hit Ratio of Caching Policies for the DB2 TPC-H Workloads

The top graph in Figure 9 shows the results for the DB2 TPC-C traces, with a server cache size of 180K pages. We obtained similar results with the DB2 TPC-C traces for other server cache sizes. In all cases, tracking the 20 most frequent hints (i.e., setting $k = 20$) was sufficient to achieve a read hit ratio close to what we could obtain by tracking all of the hints in the trace. In many cases, tracking fewer than 10 hints sufficed. The curve for the DB2_C540 trace illustrates that the Space Saving algorithm that we use to track frequent hint sets can sometimes suffer from some instability, in the sense that larger values of k may result in worse performance than smaller k . This is because hint sets reported by the Space

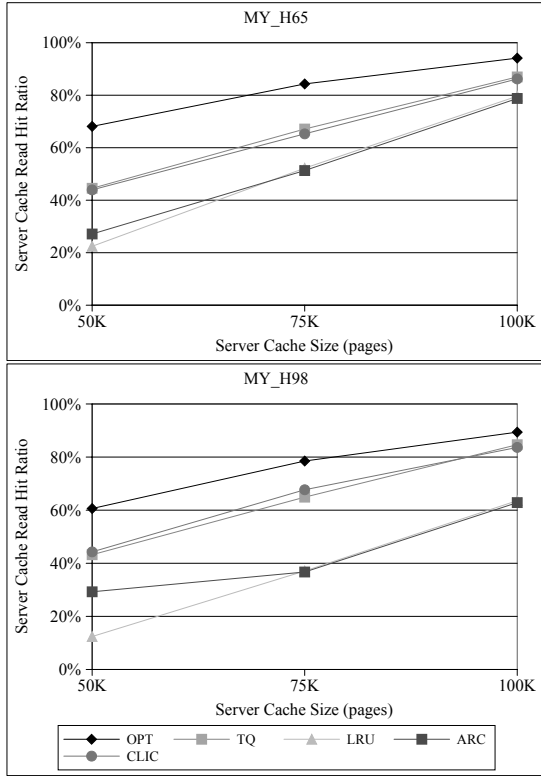


Figure 8: Read Hit Ratio of Caching Policies for the MySQL Workloads

Saving algorithm when $k = k_1$ are not guaranteed to be reported by the space saving algorithm when $k > k_1$. We only observed this problem occasionally, and only for very small values of k .

The lower graph in Figure 9 shows the results for the DB2 TPC-H traces, with a server cache size of 180K pages. For all of the DB2 TPC-H traces and all of the cache sizes that we tested, $k = 10$ was sufficient to obtain performance close to that obtained by tracking all hint sets. For the MySQL TPC-H traces (not shown in Figure 9), which contained fewer distinct hint sets, $k = 4$ was sufficient to obtain good performance. Overall, we found the top- k approach to be very effective at cutting down the number of hints to be considered by CLIC.

6.3 Increasing the Number of Hints

In the previous experiment, we studied the effectiveness of the top- k approach at reducing the number of hints that must be tracked by CLIC. In this experiment, we consider a similar question, but from a different perspective. Specifically, we consider a scenario in which CLIC is subjected to useless “noise” hints, in addition to the useful hints that it has exploited in our previous experiments. We limit the number of hint sets that CLIC is able to track and increase the level “noise”. Our objective is

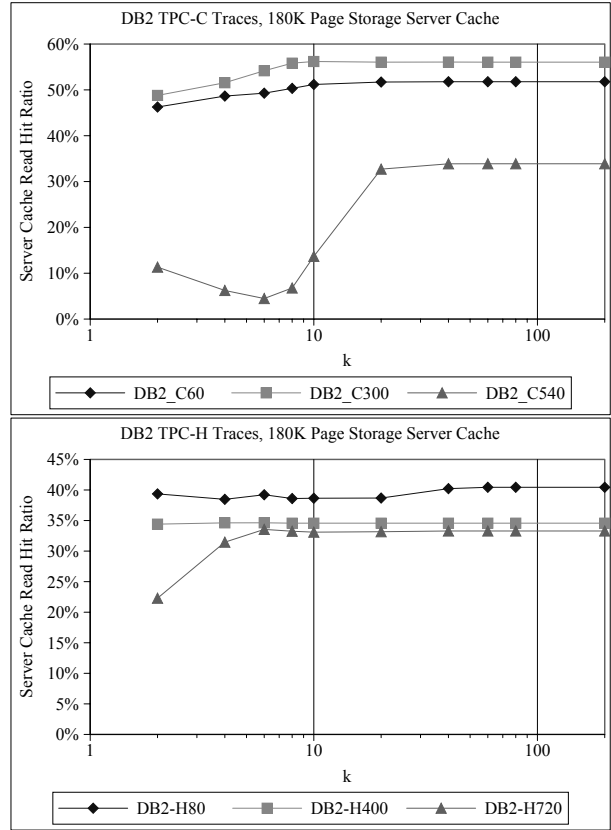


Figure 9: Effect of Top-K Hint Set Filtering on Read Hit Ratio

to determine whether the top- k approach is effective at ignoring the noise, and focusing the limited space available for hint-tracking on the most useful hints.

In practice, we hope that storage clients will not generate lots of useless hints. However, in general, clients will not be able to determine how useful their hints are to the server, and some hints generated by clients may be of little value. By deliberately introducing a controllable level of useless hints in the his experiment, we hope to test CLIC’s ability to tolerate them without losing track of those hints that are useful..

For this experiment we used our DB2 TPC-C traces, each of which contains 5 real hint types, and added T additional synthetic hint types. In other words, each request will have $5 + T$ hints associated with it, the five original hints plus T additional synthetic hints. Each injected synthetic hint is chosen randomly from a domain of D possible hint values. A particular value from the domain is selected using a Zipf distribution with skew parameter $z = 1$. When $T > 1$, each injected hint value is chosen independently of the other injected hints for the same record. Since the injected hints are chosen at random, we do not expect them to provide any informa-

tion that is useful for server cache management. This injection procedure potentially increases the number of distinct hint sets in a trace by a factor D^T . For our experiments, we chose $D = 10$, and we varied T , which controls the amount of “noise”.

Figure 10 shows the read hit ratios in a server cache of size 180K pages as a function of T . We fixed $k = 100$ for the top- k algorithm, so the number of hints tracked by CLIC remains fixed at 100 as the number of useless hints increases. As T goes from 0 to 3, the total number of distinct hint sets in each trace increases from just over 100 (the number of distinct hint sets each TPC-C trace), to about 1000 when $T = 1$, and to more than 50000 when $T = 3$.

Ideally, the server cache read hit ratio would remain unchanged as the number of “noise” hints is increased. In practice, however, this is not the case. As shown in Figure 10, CLIC fares reasonably well for the DB2_C60 trace, suffering mild degradation in performance for $T \geq 2$. However, for the other two traces, CLIC experienced more substantial degradation, particularly for $T \geq 2$. The cause of the degradation is that high-priority hint sets from the original trace get “diluted” by the additional noise hint types. For example, with $D = 10$ and $T = 2$, each original hint set is split into as many as $D^T = 100$ distinct hint sets because of the additional noise hints that appear with each request. Since CLIC has limited space for tracking hint sets, the dilution eventually overwhelms its ability to track and identify the useful hints.

This experiment suggests that it may be necessary to tune or modify CLIC to ensure that it operates well in situations in which the storage clients provide too many low-value hints. One way to address this problem is to increase k as the number of hints increases, so that CLIC is not overwhelmed by the additional hints. Controlling this tradeoff of space versus accuracy is an interesting tuning problem for CLIC. An alternative approach is to add an additional mechanism to CLIC that would allow it to group similar hint sets together, and then track reference statistics for the groups rather than the individual hint sets. We have explored one technique for doing this, based on decision trees. However, both the decision tree technique and the tuning problem are beyond the scope of this paper, and we leave them as subjects for future work.

6.4 Multiple Storage Clients

One desirable feature of CLIC is that it should be capable of accommodating hints from multiple storage clients. The clients independently send their different hints to the storage server without any coordination among themselves, and CLIC should be able to effectively prioritize the hints to get the best overall cache hit ratio.

To test this, we simulated a scenario in which multiple

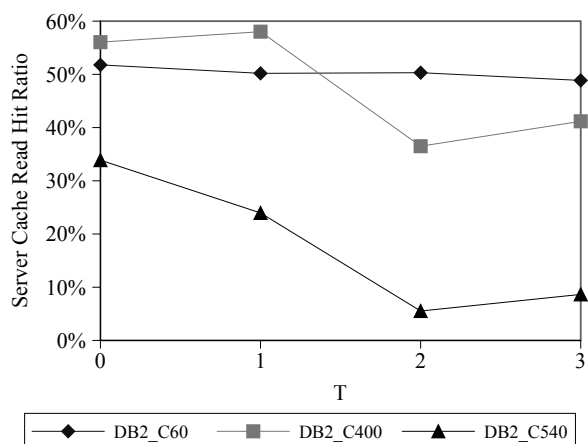


Figure 10: Effect of Number of Additional Hint Types on Read Hit Ratios

instances of DB2 share a storage server. Each DB2 instance manages its own separate database, and represents a separate storage client. All of the databases are housed in the storage server, and the storage server’s cache must be shared among the pages of the different databases. To create this scenario, we create a multi-client trace for our simulator by interleaving requests from several DB2 traces, each of which represents the requests from a single client. We interleave the requests in a round robin manner, one from each trace. We truncate all traces to the length of the shortest trace being interleaved to eliminate bias towards longer traces. We treat the hint types in each trace as distinct, so the total number of distinct hint sets in the combined trace is the sum of the number of distinct hint sets in each individual trace.

Figure 11 shows results for the trace generated by interleaving the DB2_C60, DB2_C400, and DB2_C540 traces. The server cache size is 180K pages, and CLIC uses top- k filtering with $k = 100$. The figure shows the read hit ratio for the requests from each individual trace that is part of the interleaved trace. The figure also shows the overall hit ratio for the entire interleaved trace. For comparison, the figure shows the hit ratios for the full-length (untruncated) traces when they use independent caches of size 60K pages each (i.e., the storage server cache is partitioned equally among the clients). The figure shows a dramatic improvement in hit ratio for the DB2_C60 trace and also an improvement in the overall hit ratio as compared to equally partitioning the server cache among the traces. CLIC is able to identify that the DB2_C60 trace presents the best caching opportunities (since it has the most temporal locality), and to focus on caching pages from this trace. This illustrates that CLIC is able to accommodate hints from multiple storage clients and prioritize them so as to maximize the overall

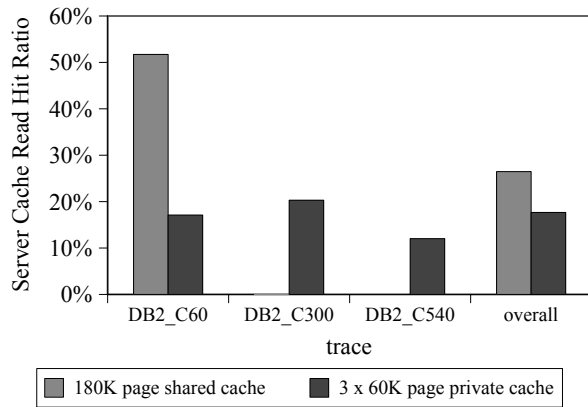


Figure 11: Read Hit Ratio with Three Clients
Read hit ratio is near zero for the DB2.C300 and DB2.C540 traces in the 180K page shared cache, so bars are not visible.

hit ratio.

Note that it is possible to consider other objectives when managing the shared server cache. For example, we may want to ensure fairness among clients or to achieve certain quality of service levels for some clients. This may be accomplished by statically or dynamically partitioning the cache space among the clients. In CLIC, the objective is simply to maximize the overall cache hit ratio without considering quality of service targets or fairness among clients. This objective results in the best utilization of the available cache space. Our experiment illustrates that CLIC is able to achieve this objective, although the benefits of the server cache may go disproportionately to some clients at the expense of others.

7 Related Work

Many replacement policies have been proposed to improve on LRU, including MQ [22], ARC [13], CAR [3], and 2Q [10]. These policies use a combination of recency of use and frequency of use to make replacement decisions. They can be used to manage a cache at any level of a cache hierarchy, though some, like MQ, were explicitly developed for use in second-tier caches, for which there is little temporal locality in the workload. ACME [1] is a mechanism that can be used to automatically and adaptively choose a good policy from among a pool of candidate policies, based on the recent performance of the candidates.

There are also caching policies that have been proposed explicitly for second (or lower) tier caches in a cache hierarchy. Chen et al [6] have classified these as either *hierarchy-aware* or *aggressively collaborative*. Hierarchy-aware methods specifically exploit the knowledge that they are running in the second tier, but they are transparent to the first tier. Some such approaches, like

X-RAY [2], work by examining the contents of requests submitted by a client application in the first tier. By assuming a particular type of client and exploiting knowledge of its behavior, X-RAY can extract client-specific semantic information from I/O requests. This information can then be used to guide caching decisions at the server. X-RAY has been proposed for file system clients [2] and DBMS clients [17].

Aggressively collaborative approaches require changes to the first tier. Examples include PROMOTE [8] and DEMOTE [19], both of which seek to maintain exclusivity among caches, and hint-based techniques, including CLIC. Although all aggressively collaborative techniques require changes to the first tier, they vary considerably in the intrusiveness of the changes that are required. For example, ULC [9] gives complete responsibility for management of the second tier cache to the first tier. PROMOTE [8] prescribes a replacement policy that must be used by all tiers, including the first tier. This may be undesirable if the first tier cache is managed by a database system or other application which prefers an application-specific policy for cache management. Among the aggressively collaborative techniques, hint-based approaches like CLIC are arguably the least intrusive and least costly. Hints are small and can be piggybacked onto I/O requests. More importantly, hint-based techniques do not require any changes to the policies used to manage the first tier caches.

Several hint-based techniques have been proposed, including importance hints [6] and write hints [11], which have already been described. In their work on informed prefetching and caching, Patterson et al [16] distinguished hints that disclose from hints that advise, and advocated the former. Most subsequent hint-based techniques, including CLIC, use hints that disclose. Informed prefetching and caching rely on hints that disclose sequential access to entire files or to portions of files. Karma [21] relies on application hints to group pages into “ranges”, and to associate an expected access pattern with each range. Unlike CLIC, all of these techniques are they are designed to exploit specific types of hints. As was discussed in Section 1, this makes them difficult to generalize and combine.

A previous study [6] suggested that aggressively collaborative approaches provided little benefit beyond that of hierarchy-aware approaches and thus, that the loss of transparency implied by collaborative approaches was not worthwhile. However, that study only considered one ad hoc hint-based technique. Li et al [11] found that the hint-based TQ algorithm could provide substantial performance improvements in comparison to hint-oblivious approaches (LRU and MQ) as well as simple hint-aware extensions of those approaches.

There has also been work on the problem of sharing a cache among multiple competing client applications [5, 12, 18, 20]. Often, the goal of these techniques is to achieve specific quality-of-service objectives for the client applications, and the method used is to somehow partition the shared cache. This work is largely orthogonal to CLIC, in the sense that CLIC can be used, like any other replacement algorithm, to manage the cache contents in each partition. CLIC can also be used to directly control a shared cache, as in Section 6.4, but it does not include any mechanism for enforcing quality-of-service requirements or fairness requirements among the competing clients.

The problem of identifying frequently-occurring items in a data stream occurs in many situations. Metwally et al [14] classify solutions to the frequent-item problem as counter-based techniques or sketch-based techniques. The former maintain counters for certain individual items, while the latter collect information about aggregations of items. For CLIC, we have chosen to use the Space-Saving algorithm [14] as it is both effective and simple to implement. A recent study [7] found the Space-Saving algorithm to be one of the best overall performers among frequent-item algorithms.

8 Conclusion

We have presented CLIC, a technique for managing a storage server cache based on hints from storage client applications. CLIC provides a general, adaptive mechanism for incorporating application-provided hints into cache management. We used trace-driven simulation to evaluate CLIC, and found that it was effective at learning to exploit hints. In our tests, CLIC learned to perform as well as or better than TQ, an ad hoc hint based technique. In many scenarios, CLIC also performed substantially better than hint-oblivious techniques such as LRU and ARC. Our results also show that CLIC, unlike TQ and other ad hoc techniques, can accommodate hints from multiple client applications.

A potential drawback of CLIC is the space overhead that is required learning which hints are valuable. We considered a simple technique for limiting this overhead, which involves identifying frequently-occurring hints and tracking statistics only for those hints. In many cases, we found that it was possible to significantly reduce the number of hints that CLIC had to track with only minor degradation in performance. However, although tracking only frequent hints is a good way to reduce overhead, the overhead is not eliminated and the space required for good performance may increase with the number of hint types that CLIC encounters. As part of our future work, we are using decision trees to generalize hint sets by grouping related hint sets together into

a common class. We expect that this approach, together with the frequency-based approach, can enable CLIC to accommodate a large number of hint types.

9 Acknowledgements

We are grateful to Martin Kersten and his group at CWI for their assistance in setting up TPC-H on MySQL, and to Aamer Sachedina and Roger Zheng at IBM for their help with the DB2 modifications and trace collection. Thanks also to our shepherd, Liuba Shrira, and the anonymous referees for their comments and suggestions. This work was supported by the Ontario Centre of Excellence for Communication and Information Technology and by the Natural Sciences and Engineering Research Council of Canada.

References

- [1] ARI, I., AMER, A., GRAMACY, R., MILLER, E. L., BRANDT, S., AND LONG, D. D. E. ACME: Adaptive caching using multiple experts. In *Workshop on Distributed Data and Structures 4 (WDAS)* (Mar. 2002), Carleton Scientific, pp. 143–158.
- [2] BAIRAVASUNDARAM, L. N., SIVATHANU, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. X-RAY: A non-invasive exclusive caching mechanism for RAID5. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)* (June 2004).
- [3] BANSAL, S., AND MODHA, D. CAR: Clock with adaptive replacement. In *Proc. of the 3rd USENIX Symposium on File and Storage Technologies (FAST'04)* (Mar. 2004).
- [4] BELADY, L. A. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101.
- [5] BROWN, K. P., CAREY, M. J., AND LIVNY, M. Goal-oriented buffer management revisited. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (June 1996), pp. 353–364.
- [6] CHEN, Z., ZHANG, Y., ZHOU, Y., SCOTT, H., AND SCHIEFER, B. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05)* (2005), pp. 145–156.

- [7] CORMODE, G., AND HADJIELEFTHARIOU, M. Finding frequent items in data streams. In *Proc. Int'l Conference on Very Large Data Bases (VLDB'08)* (Aug. 2008).
- [8] GILL, B. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *Proc. USENIX Conference on File and Storage Technologies (FAST'08)* (2008), pp. 49–65.
- [9] JIANG, S., AND ZHANG, X. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *Proc. 24th International Conference on Distributed Computing Systems (ICDCS'04)* (2004), pp. 168–177.
- [10] JOHNSON, T., AND SHASHA, D. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. International Conference on Very Large Data Bases (VLDB'94)* (1994), pp. 439–450.
- [11] LI, X., ABOULNAGA, A., SALEM, K., SACHEDINA, A., AND GAO, S. Second-tier cache management using write hints. In *USENIX Conference on File and Storage Technologies (FAST'05)* (Dec. 2005), pp. 115–128.
- [12] MARTIN, P., LI, H.-Y., ZHENG, M., ROMANUFA, K., AND POWLEY, W. Dynamic reconfiguration algorithm: Dynamically tuning multiple buffer pools. In *11th International Conference on Database and Expert Systems Applications (DEXA)* (2000), pp. 92–101.
- [13] MEGIDDO, N., AND MODHA, D. S. ARC: A self-tuning, low overhead replacement cache. In *Proc. USENIX Conference on File and Storage Technology (FAST'03)* (2003).
- [14] METWALLY, A., AGRAWAL, D., AND ABBADI, A. E. Efficient computation of frequent and top-k elements in data streams. In *Proc. International Conference on Database Theory (ICDT)* (Jan. 2005).
- [15] MUNTZ, D., AND HONEYMAN, P. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. In *Proceedings of the USENIX Winter Conference* (Jan. 1992), pp. 305–313.
- [16] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proc. ACM Symposium on Operating Systems Principles (SOSP'95)* (Dec. 1995), pp. 79–95.
- [17] SIVATHANU, M., BAIRAVASUNDARAM, L. N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Database-aware semantically-smart storage. In *Proc. of the USENIX Symposium on File and Storage Technologies (FAST'05)* (2005), pp. 239–252.
- [18] SOUNDARARAJAN, G., CHEN, J., SHARAF, M., AND AMZA, C. Dynamic partitioning of the cache hierarchy in shared data centers. In *Proc. Int'l Conference on Very Large Data Bases (VLDB'08)* (Aug. 2008).
- [19] WONG, T. M., AND WILKES, J. My cache or yours? making storage more exclusive. In *USENIX Annual Technical Conference (USENIX 2002)* (June 2002), pp. 161–175.
- [20] YADGAR, G., FACTOR, M., LI, K., AND SCHUSTER, A. Mc2: Multiple clients on a multilevel cache. In *Proc. Int'l Conference on Distributed Computing Systems (ICDCS'08)* (June 2008).
- [21] YADGAR, G., FACTOR, M., AND SCHUSTER, A. Karma: Know-it-all replacement for a multilevel cache. In *Proc. USENIX Conference on File and Storage Technologies (FAST'07)* (Feb. 2007).
- [22] ZHOU, Y., CHEN, Z., AND LI, K. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems* 15, 7 (July 2004).

Notes

¹DB2 Universal Database is a registered trademark of IBM.

²MySQL is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Minuet: Rethinking Concurrency Control in Storage Area Networks

Andrey Ermolinskiy[†], Daekyeong Moon[†], Byung-Gon Chun^{*}, Scott Shenker[†] [‡]

[†]University of California at Berkeley, ^{*}Intel Research Berkeley, [‡]ICSI

Abstract

Clustered applications in storage area networks (SANs), widely adopted in enterprise datacenters, have traditionally relied on distributed locking protocols to coordinate concurrent access to shared storage devices. We examine the semantics of traditional lock services for SAN environments and ask whether they are sufficient to guarantee data safety at the application level. We argue that a traditional lock service design that enforces strict *mutual exclusion* via a *globally-consistent view of locking state* is neither sufficient nor strictly necessary to ensure application-level correctness in the presence of asynchrony and failures. We also argue that in many cases, strongly-consistent locking imposes an additional and unnecessary constraint on application availability. Armed with these observations, we develop a set of novel concurrency control and recovery protocols for clustered SAN applications that achieve safety and liveness in the face of arbitrary asynchrony, crash failures, and network partitions. Finally, we present and evaluate Minuet- a new synchronization primitive based on these protocols that can serve as a foundational building block for safe and highly-available SAN applications.

1 Introduction

In recent years, storage area networks (SANs) have been gaining widespread adoption in enterprise datacenters [19] and are proving effective in supporting a range of applications across a broad spectrum of industries. According to a recent survey of IT professionals across a range of corporations, government agencies, and universities, the overwhelming majority (80%) have deployed a storage area network in their organizations and 26% of the respondents report having deployed five or more SANs [14]. Some of the common applications include online transaction processing in finance and e-commerce, digital media production, business data analytics, and high-performance scientific computing.

A SAN architecture is a particularly attractive choice for parallel clustered applications that demand high-speed concurrent access to a scalable storage backend. Such applications commonly rely on a clustered middleware service to provide a higher-level storage abstraction such as a filesystem (GFS [35], OCFS [8], PanFS [10], GPFS [37]) or a relational database (Oracle RAC [9]) on top of raw disk blocks.

One of the primary design challenges for clustered SAN applications and middleware is ensuring safe and efficient coordination of access to application state and metadata that resides on shared storage. The traditional approach to concurrency control in shared-disk clusters involves the use of a synchronization module called a *distributed lock manager* (DLM). Typically, DML services aim to provide the guarantee of *strict mutual exclusion*, ensuring that no two processes in the system can simultaneously hold conflicting locks. In abstract terms, providing such guarantees requires enforcing a globally-consistent view of lock acquisition state and one could argue that a traditional DLM design views such consistency as an end-in-itself rather than a means to achieving application-level correctness.

In this paper, we take a close look at the semantics of SAN lock services and ask whether the assurances of full mutual exclusion and strongly-consistent locking are, in fact, a prerequisite for correct application behavior. Our main finding is that the standard semantics of mutual exclusion provided by a DLM are neither strictly necessary nor sufficient to guarantee safe coordination in the presence of node failures and asynchrony. In particular, processing and queuing delays in SAN switches and host bus adapters (HBAs) expose applications to out-of-order delivery of I/O requests from presumed faulty processes which, in certain scenarios, can incur catastrophic violations of safety and cause permanent data loss.

We propose and evaluate a new technique for disk access coordination in SAN environments. Our approach augments target storage devices with a tiny application-independent functional component, called a *guard*, and a small amount of state, which enable them to reject inconsistent I/O requests and provide a property called *session isolation*.

These extensions enable a novel *optimistic* approach to concurrency control in SANs and can also make existing lock-based protocols safe in the face of arbitrarily delayed message delivery, drifting clocks, crash process failures, and network partitions. The session isolation property in turn provides a foundational primitive for implementing more complex and useful coordination semantics, such as *serializable transactions*, and we demonstrate one such protocol.

We then describe the implementation of Minuet- a software library that provides a novel synchronization

primitive for SAN applications based on the protocols we present. Minuet assumes the presence of guard at the target storage devices and provides applications with locking and distributed transaction facilities, while guaranteeing liveness and data safety in the face of arbitrary asynchrony, node failures, and network partitions. Our evaluation shows that applications built atop Minuet compare favorably to those that rely on a conventional strongly-consistent DLM, offering comparable or better performance and improved availability.

Unlike existing services for fault-tolerant distributed coordination such as Chubby [20] and Zookeeper [15], Minuet requires its lock managers to maintain only loosely-consistent replicas of locking state and thus permits applications to make progress with less than a majority of lock manager replicas. To demonstrate the practical feasibility of our approach, we implemented two sample applications – a distributed chunkmap and a B+ tree – on top of Minuet and evaluated them in a clustered environment supported by an iSCSI-based SAN.

The benefits of optimistic concurrency control and the associated tradeoffs have been extensively explored in the database literature and are well understood. In particular, techniques such as callback locking, optimistic 2-phase locking, and adaptive callback locking [18, 21, 24, 42] have been proposed to enable safe coordination and efficient caching in client-server databases. It is important to note, however, that these approaches are not directly applicable to SANs because they assume the existence of a central lock server, typically co-located with the data block storage server. This assumption does not hold in a SAN environment, where the storage "servers" are application-agnostic disk arrays that possess no knowledge of locking state or node liveness status. Hence, a conservative DLM service that enforces strict mutual exclusion has traditionally been viewed as the only practical method of coordinating concurrent access to shared state for SAN applications.

Our main insight is that a single nearly trivial extension to the internal logic of a SAN storage device suffices to address the data safety problems associated with traditional DLMs and enables a very different approach to protocol layering for storage access coordination. Crucially, we achieve this without introducing application-level logic into storage devices and without forfeiting the generality and simplicity of the traditional block-level interface to SAN-attached devices.

The technical feasibility of device-based synchronization and its practical advantages have been demonstrated by several earlier proposals [12, 29]. Our study builds on this earlier work and while prior efforts have primarily focused on moving the functionality of a traditional cluster lock manager into the storage device, Minuet aims to provide a more general and useful synchronization prim-

itive that supports a wider range of concurrency control mechanisms. In addition to supporting traditional *conservative* locking, our approach enables an *optimistic* method of concurrency control that can improve performance for certain application workloads. Further, Minuet allows existing locking protocols to remain safe in the presence of arbitrarily-delayed message delivery, node failures, and network partitions.

The rest of this paper is organized as follows. In Section 2, we provide the relevant background on SAN and some representative examples of data safety problems. In Section 3, we present our main contribution – the design of Minuet, a novel safe and highly available synchronization mechanism for SAN applications. Section 4 describes our prototype implementation and two sample parallel applications. We evaluate our system in Section 5 and discuss practical aspects of our approach in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 Background

2.1 Storage area networks

Storage area networks (SANs) are popular in enterprise datacenters and are commonly adopted to support the storage needs of data-intensive clustered applications. In the SAN (or *shared-disk*) model, persistent storage devices, typically disk drive arrays or specialized hardware appliances, are attached to a dedicated *storage network* and appear to members of the application cluster as local disks. Most SANs utilize a combination of SCSI and a low-level transport protocol such as TCP/IP or FCP (Fibre Channel Protocol) for communication between the application nodes and the target storage devices.

SANs aim to provide fully decentralized access to shared application state on disk and in principle, any SAN-attached client node can access any piece of data without routing its requests to a dedicated server. While in this model, all requests on a particular piece of data are centrally serialized, the crucial distinction from the traditional *server-attached* storage paradigm is that the point of serialization is a hardware disk controller that exposes an application-independent I/O interface on raw disk blocks and is oblivious to application semantics and data layout considerations.

Broadly, the SAN paradigm is advantageous from the standpoint of availability because it offers better redundancy and decouples node failures from loss of persistent state. Incoming application requests can be routed to any available node in the cluster and, in the event of a node failure, subsequent requests can be redirected to another processor with minimal interruption of service.

One of the primary design challenges for clustered SAN applications and middleware is ensuring safe and

efficient coordination of access to shared state on disk and commonly, a software service called a *distributed lock manager* (DLM) is employed to provide such coordination. A typical lock service such as OpenDLM [7] operates on *shared resources*, abstract application-level entities that require access coordination, and attempts to provide the guarantee of *mutual exclusion* - no two processes may simultaneously hold conflicting locks on the same resource.

2.2 Safety and liveness problems in SANs

In theory, DLM-based mutual exclusion offers sufficient mechanism to ensure safe access to shared state. In practice, however, guaranteeing safe serialization of disk requests tends to be more difficult than the above discussion might suggest due to the effects of *node failures* and *asynchrony*: nodes can fail by stopping and the processing and communication delays are not bounded. The following examples illustrate the nature of the problem.

Scenario 1: Consider a data structure S spanning 10 blocks on a shared disk D and two clients, C_1 and C_2 , that are accessing the data structure concurrently. C_1 is updating blocks [3 – 7] of S under the protection of an exclusive lock, while C_2 wants to read S in its entirety (i.e., blocks [0 – 9]) and is waiting for a shared lock. Suppose C_1 crashes after sending its *WRITE* request to D but before hearing the response. The lock manager correctly detects the failure, reclaims the exclusive lock, and grants it to C_2 in shared mode. Next, C_2 proceeds to reading S and, assuming that a single disk request can carry up to 5 blocks of data, issues two requests: $R_1 = \langle \text{READ}[0 - 4] \rangle$ and $R_2 = \langle \text{READ}[5 - 9] \rangle$. Suppose C_1 's delayed *WRITE* request on blocks [3 – 7] reaches the disk after R_1 but before R_2 , in which case only the latter would reflect the effects of C_1 's update. Hence, although individual I/O requests are processed by D as atomic units, their inconsistent interleaving would cause C_2 to observe and act upon a *partial update* from C_1 , which can be viewed as a violation of data safety.

As an alternative to heartbeat failure detection, a lease-based mechanism [26] can be used to coordinate clients' accesses in the above example, but precisely the same problematic scenario would arise when clocks are not synchronized. When C_1 crashes and its lease expires, the lease manager could grant it to C_2 prior to the arrival of the last *WRITE* from C_1 to the storage target. Since the target does not coordinate with the lease manager, it fails to establish the fact that an incoming request from C_1 is inconsistent with the current lease ownership state.

Scenario 2: Clustered applications and middleware services commonly need to enforce transactional semantics on updates to application state and metadata. In a shared-disk clustered environment, distributed transactions have traditionally been supported by two-phase

locking in conjunction with a distributed write-ahead logging (WAL) protocol. In the abstract, the system maintains a snapshot of application state along with a set of per-client logs (also on shared disks) that record *Redo* and/or *Undo* information for every transaction along with its commit status. During failure recovery, the system must examine the suspected client's log and restore consistency by rolling back all uncommitted updates and re-playing all updates associated with committed transactions that may not have been flushed to the snapshot prior to the failure. An essential underlying assumption here is that once log recovery is initiated, no additional *WRITE* requests from the suspected process will reach the snapshot. A violation of this assumption could result in the corruption of logs and application data.

Ensuring data safety in a shared-disk environment has traditionally required a set of *partial synchrony assumptions* to allow reliable heartbeat-driven failure detection and/or leases. For example, lease-based mechanisms typically expect bounded clock drift rates and message delivery delays to ensure the absence of in-flight I/O requests upon lease termination. However, these assumptions are probabilistic at best and since application data integrity is predicated on the validity of these assumptions, failure timeouts must be tuned to a very conservative value to account for worst-case delays in switch queues and client-side buffering. Such (necessarily) pessimistic timeouts may have a profoundly negative impact on failure recovery times - one of the common criticisms of SAN-oriented applications [16].

Another serious limitation exhibited by today's SAN applications is *liveness*. The DLM (or lease manager) represents an additional point of failure and while various fault tolerance techniques can be applied to improve its availability, the very nature of the semantics enforced by the DLM places a fundamental constraint on the overall system availability. For instance, multiple lock manager replicas can be deployed in a cluster, but mutual exclusion can be guaranteed only if clients' requests are presented to them in a consistent order, which necessitates consensus mechanisms such as Paxos [31]. Alternatively, a single lock manager instance can be elected dynamically [27] from a group of candidates and in this case, ensuring mutual exclusion necessitates global agreement on the lock manager's identity. In both cases, reaching agreement fundamentally requires access to an active primary component - typically a majority of nodes. As a result, a large-scale node failure or a network partition that renders the primary component unavailable or unreachable may bring about a system-wide outage and complete loss of service.

To summarize, today's SAN applications and middleware face significant limitations along the dimensions of safety and liveness. At present, several hardware-

assisted techniques, such as out-of-band power management (STOMITH) [3], SAN fabric fencing [1], and SCSI-3 PR [11] can be employed to mitigate some of these issues. These mechanisms help reduce the likelihood of data corruption under common failure scenarios, but do not provide the desired assurances of safety and liveness in the general case and, as we would argue, do not address the underlying problem. We observe that the underlying problem may be a case of *capability mismatch* between "intelligent" application processes that possess full knowledge of application's data structures, their disk layout, and consistency semantics on the one hand and relatively "dumb" storage devices on the other. The safety and liveness problems illustrated above can be attributed to a disk controller's inability to identify and appropriately react to the various application-level events such as *lock release*, *failure suspicion*, and *failure recovery action*.

3 Minuet Design

At a high level, our approach reexamines the correctness criteria that a cluster DLM service must provide to applications. Traditionally, DLMs tend to treat shared application resources as purely abstract entities and enforce the *mutual exclusion* property: no two clients may simultaneously hold conflicting locks on the same shared resource. We note, however, that the mutual exclusion property as stated above is provably unattainable in an asynchronous system that is subject to even a single crash failure - a consequence of the impossibility of consensus [23] in such an environment. Furthermore, as we explain in the previous section, a hypothetical lock service that does offer such guarantees would not by itself suffice to guarantee data safety in such a setting due to the possibility of out-of-order I/O request delivery.

Rather than restricting access to critical code sections, our approach views the access coordination problem in terms of I/O request ordering guarantees that the storage system must provide to application processes. We refer to this alternate notion of correctness as *session isolation*.

We define this correctness property in formal terms below and then present a protocol that achieves session isolation with the help of guard logic. Finally, we demonstrate how distributed multi-resource transactions can be supported using session isolation as a building block.

3.1 Session isolation

Throughout this paper, we will use the term *resource* to denote the basic logical unit of concurrency control. Each resource R is identified by a unique and persistent application-level identifier (denoted $R.resID$) and has some physical representation on a SAN-attached storage device, which we call its *owner* ($R.owner$). More concretely, a resource may represent a filesystem block, a

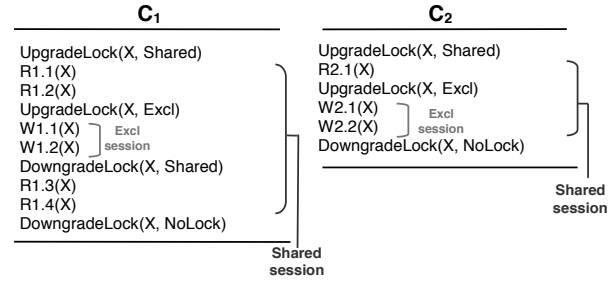


Figure 1: Concurrent request streams to a shared resource X from two client processes, C_1 and C_2 . In this example, C_1 first performs two *READ* operations on X under the protection of a *Shared* lock, then upgrades to *Excl* and issues two *WRITES*. Lastly, C_1 downgrades its lock to *Shared* and performs two more *READs*. Client C_2 acquires a *Shared* lock on X and submits a *READ* request, followed by an upgrade to *Excl* and two *WRITE* requests.

database table, or an individual tuple in a table. An application process operates on R by issuing *READ/WRITE* commands to $R.owner$, as well as by acquiring and releasing locks on $R.resID$. We begin by defining the notion of *session* to a shared resource and describing the *session isolation* criterion.

Definition 1. If a client process C requests a *Shared* lock on R and the request is granted by the lock service, we say that C establishes a **shared session** to R . An existing shared session is terminated when C releases the *Shared* lock (i.e., downgrades to *None*). Analogously, by acquiring an *Excl* lock, a client establishes an **exclusive session** to R that can subsequently be terminated by downgrading to *Shared* or *None*.

We define $Sessions(T, C, R)$ to be the set of all sessions to R from C active at time T , which is determined solely by the sequence of C 's prior upgrade and downgrade requests to the lock service. $Sessions(T, C, R)$ may contain a shared or an exclusive session to R , or both, or none.

We say that a shared session **conflicts** with every exclusive session to the same resource R and an exclusive session **conflicts** with every other session to R .

Definition 2. If a client process C issues at time T a disk request r that operates on shared resource R , we say that r **belongs to** session S if $S \in Sessions(T, C, R)$. For a given session S , we additionally define $Requests(S)$ to be the set of all disk requests that belong to S .

Definition 3. A given global execution history satisfies **session isolation** with respect to R if the sequence of disk request messages $M = \langle r_1, r_2, \dots \rangle$ observed and processed in this history by $R.owner$ satisfies: $\forall r_i, r_j \in M$ such that $\{r_i, r_j\} \subset Requests(S)$ for some $S : \nexists r_k \in M$ such that $i < k < j$ and $r_k \in Requests(S^*)$ for a session S^* from another client that conflicts with S .

Informally, the above condition requires $R.owner$ to observe the prefixes of all sessions to R in strictly se-

rial order, ensuring that no two requests in a client's session are interleaved by a conflicting request from another client. To illustrate this definition, consider a pair of concurrent request sequences shown in Figure 1. In this scenario, the following two orderings of request observations by the owner of shared resource X would satisfy session isolation:

$$E_1 = \langle R_{1.1}, R_{1.2}, W_{1.1}, W_{1.2}, R_{1.3}, R_{1.4}, R_{2.1}, W_{2.1}, W_{2.2} \rangle$$

$$E_2 = \langle R_{1.1}, R_{1.2}, W_{1.1}, R_{2.1}, W_{2.1}, W_{2.2} \rangle$$

However, an execution history that causes the owner to observe $\langle R_{1.1}, R_{2.1}, R_{1.2}, W_{1.1}, W_{2.1} \rangle$ does not obey session isolation because it permits $R_{2.1}$ and $W_{2.1}$, two shared-session requests from C_2 , to be interleaved by $W_{1.1}$, an exclusive-session request from C_1 .

Note that session isolation is more permissive than strict mutual exclusion and in particular, permits execution histories in which two clients simultaneously hold conflicting locks on the same shared resource. At the same time, one could argue that these semantics meaningfully capture the essence of shared-disk locking, by which we mean that the request ordering guarantees provided by session isolation are precisely those that applications developers have come to expect from a traditional DLM. To see this, observe that in the previous example, a conventional lock service offering full mutual exclusion would cause X to observe E_1 by granting clients' requests in the order $\langle C_1(Shared), C_1(Excl), C_2(Shared), C_2(Excl) \rangle$. Likewise, E_2 corresponds to a possible failure scenario in which C_1 crashes after acquiring its locks, causing the DLM to reclaim them and grant ownership to C_2 .

3.2 Guard

Our core approach is inspired by earlier work on bridging the intelligence gap between applications and block storage devices [17, 25], as well as earlier proposals for device-based synchronization [12, 29]. We augment SAN-attached disks with a small application-independent component, which we call a *guard*, that enforces the session isolation invariant on the stream of incoming I/O commands. We associate a *session identifier* (*SID*) with every client session to a shared resource and modify the storage protocol stack on the initiators to annotate all outgoing disk commands with the current *SID* for the respective resource. Below, we refer to this additional state in the command header as *session annotation*.

A session annotation for a disk command operating on R has two components: a *session verifier* and a *session update*, denoted by $R.verifySID$ and $R.updateSID$, respectively. For commands that belong to an existing session, the verifier enables the target to confirm session validity prior to accepting the command and *updateSID* is used by the initiator to signal the start of a new session.

For each shared resource R , its owner device maintains a local session identifier (denoted $R.ownerSID$) on persistent storage. Upon receipt of an I/O command from an initiator, the owner invokes the guard, which evaluates the command's session annotation against $R.ownerSID$ and determines whether session isolation would be preserved by accepting the command. Functionally, the guard operation is a form of compare-and-set and we describe this operation in detail in Section 3.3.

If an incoming I/O request fails verification, the target drops the request from its input queue and notifies the initiator via a special status code *EBADSESSION*. From an application developer's point of view, session rejection appears as a failed disk request along with an exception notification from the lock service indicating that a lock on the respective resource is no longer valid.

The guard situated at the target devices addresses the safety problems due to delayed messages and inconsistent failure observations that plague asynchronous distributed environments and enforcing safety at the target device permits us to simplify the core functionality of the DLM module. In our scheme, the primary purpose of the lock service is ensuring an efficient assignment of session identifiers to clients that minimizes the frequency of command rejection for a given application workload.

Decoupling correctness from performance in this manner enables substantial flexibility in the choice of mechanism used to control the assignment of session identifiers. At one extreme is a purely optimistic technique, whereby every client selects its *SIDs* via an independent local decision without attempting to coordinate with the rest of the cluster and this might be an entirely reasonable strategy for applications and workloads characterized by a consistently low rate of data contention. A traditional DLM service that serializes all session requests at a central lock server can be viewed as a design point at the other extreme. Minuet aims to position itself in the continuum between these extremes and allow application developers to trade off lock service availability, synchronization overhead, and I/O performance.

3.3 Enforcing session isolation

Minuet uses a simple timestamp-based mechanism to guarantee the session isolation invariant. A client's session to a given resource R is identified by a value pair $\langle T_s, T_x \rangle$ specifying a *shared* and an *exclusive* timestamp, respectively. These timestamps are globally unique - no two sessions from distinct clients are identified using the same pair of values and no client is assigned the same value pair twice. To ensure global uniqueness, we use the following timestamp format: $\langle T.incNum.cntID \rangle$, where *cntID* uniquely identifies the client process and *incNum* is the client's *incarnation number* - a monotonic counter ensuring uniqueness across crashes.

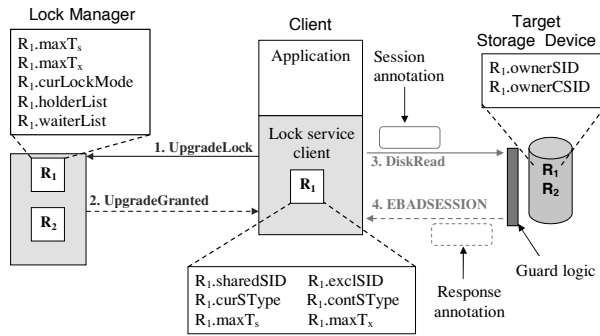


Figure 2: Protocol messages and per-resource state at application clients, lock managers, and shared storage devices.

Client-side state: For each shared resource R , a client C maintains a pair of session identifiers for its shared and exclusive sessions to R , denoted by $R.sharedSID$ and $R.exclSID$, respectively. Additionally, $R.curSType$ identifies the *current session type*, one of $\{None, Shared, Excl\}$, and $R.contSType$ holds the client's *session continuation type*. The latter value is used by the target device to verify (prior to executing a request from C) that its existing session has not been broken by a conflicting request from another client. Finally, every client C maintains an estimate of the largest shared and exclusive timestamp values previously assigned to a session identifier for any client, which we denote by $R.maxTs$ and $R.maxTx$. Initially, $R.sharedSID = R.exclSID = NIL$, $R.curSType = R.contSType = None$, and $R.maxTs = R.maxTx = 0$. The steps and states of the basic locking and storage access protocols are illustrated in Figure 2.

Acquiring locks: To acquire/upgrade a lock on resource R , a client C proposes a unique session timestamp pair $\langle proposedTs, proposedTx \rangle$ to the lock manager. To acquire a *Shared* lock on R , C sets $proposedTx \leftarrow R.maxTx$ and sets $proposedTs$ to some unique timestamp greater than $R.maxTs$. The client then sends an *UpgradeLock* request to the lock manager, specifying the desired mode (*Shared*) and the proposed timestamp pair. The lock manager accepts and enqueues this request if no request with a larger $proposedTx$ value has been accepted. Otherwise, the manager denies the request and responds with *UpgradeDenied*, which includes the largest timestamp values observed by the manager. In the latter case, the client updates its local estimates $\langle R.maxTs, R.maxTx \rangle$ and submits a new proposal. After accepting and enqueueing C 's request, the lock manager eventually grants it and responds with *UpgradeGranted*. The client then sets $R.curSType \leftarrow Shared$ and initializes the shared session identifier: $R.sharedSID \leftarrow \langle proposedTs, proposedTx \rangle$.

To upgrade a lock from *Shared* to *Excl*, the client sends *UpgradeLock* to the lock manager after setting $proposedTs \leftarrow R.maxTs$ and $proposedTx$ to some

Setting up a session annotation during I/O request submission:

```

if ( $R.curSType = Shared$ ) /* Shared session is active */
   $R.updateSID \leftarrow R.sharedSID$ ;
   $R.verifySID.Ts \leftarrow NIL$ ;  $R.verifySID.Tx \leftarrow R.sharedSID.Tx$ ;
else /* Exclusive session is active */
   $R.updateSID \leftarrow R.exclSID$ ;
  if ( $R.contSType = Shared$ )
     $R.verifySID.Ts \leftarrow NIL$ ;  $R.verifySID.Tx \leftarrow R.sharedSID.Tx$ ;
  else
     $R.verifySID \leftarrow R.exclSID$ ;

```

Guard logic at the target device:

```

Use the resource identifier ( $R.resID$ ) to look up  $R.ownerSID$ ;
if ( $R.verifySID.Tx < R.ownerSID.Tx$ ) REJECT;
if ( $R.verifySID.Ts \neq NIL$ )
  if ( $R.verifySID.Ts < R.ownerSID.Ts$ ) REJECT;
if (REJECTED)
  Respond to the initiator with  $\langle EBADSESSION, R.ownerSID \rangle$ ;
else
   $R.ownerSID.Ts \leftarrow \max(R.ownerSID.Ts, R.updateSID.Ts)$ ;
   $R.ownerSID.Tx \leftarrow \max(R.ownerSID.Tx, R.updateSID.Tx)$ ;
  Execute the command and respond to the initiator;

```

Figure 3: Disk request submission and guard logic pseudocode.

unique timestamp greater than $R.maxTx$. Upon receiving *UpgradeGranted* from the lock manager, the client sets $R.curSType \leftarrow Excl$ and $R.exclSID \leftarrow \langle proposedTs, proposedTx \rangle$. Upgrading from *None* to *Excl* is functionally equivalent to acquiring a *Shared* lock and then upgrading to *Excl*, but as an optimization, these operations can be combined into a single request to the lock manager.

Accessing shared storage: After establishing a session to R by acquiring a corresponding lock, client can proceed to issuing disk requests that operate on the content of R . Each outgoing request is augmented with a session annotation that enables the target device to verify proper ordering of requests and enforce session isolation. The annotation carries a tuple of the form $\langle R.resID, R.verifySID, R.updateSID \rangle$ and is initialized as shown in Figure 3.

Upon receipt of a disk request from a client, the owner device invokes the guard logic, which evaluates the session annotation as specified in Figure 3. In the event of rejection, the owner immediately discards the command and sends an *EBADSESSION* response to the client, together with a *response annotation* carrying $\langle R.ownerSID \rangle$. Otherwise, the owner executes (or enqueues) the command and updates its local session identifier as shown in the figure.

Upon receipt of an *EBADSESSION* status code, the initiator examines the response annotation and notifies the application process that its lock and session on R is no longer valid. The condition $R.verifySID.Ts < R.ownerSID.Ts$ indicates interruption of an exclusive session, in which case the client downgrades its lock to *Shared*, sets $\langle R.curSType, R.contSType \rangle \leftarrow Shared$, and sets $R.exclSID \leftarrow NIL$. A *Shared* lock is further downgraded to *None* if $R.verifySID.Tx < R.ownerSID.Tx$.

(since in this case, a conflicting exclusive-session request has been accepted). In this situation, the client sets $\langle R.curSType, R.contSType \rangle \leftarrow None$ and $R.sharedSID \leftarrow NIL$. In both cases, the maximum timestamp estimates $R.maxT_s$ and $R.maxT_x$ are updated to reflect the most recent timestamps observed by the owner.

Upon receiving a *SUCCESS* status code, the client sets $R.contSType \leftarrow R.curSType$ and updates the shared session identifier to reflect the most recent value in the annotation: $R.sharedSID \leftarrow R.updateSID$. (This step is necessary to ensure that a shared session remains valid after a *Shared* \rightarrow *Excl* upgrade or a downgrade to *Shared*).

Downgrading locks: To downgrade an existing lock from *Excl* to *Shared*, the client sends a *DowngradeLock* request to the lock manager and resets the exclusive-session state: $R.exclSID \leftarrow NIL$, $\langle R.curSType, R.contSType \rangle \leftarrow Shared$. Similarly, to downgrade from *Shared* to *None*, the client notifies the lock manager and sets $R.sharedSID \leftarrow NIL$, $\langle R.curSType, R.contSType \rangle \leftarrow None$. Upon receipt of a *DowngradeLock* request, the manager updates the ownership state for R and, if possible, grants the lock to the next waiter in the queue.

Correctness: The locking protocol and the guard described above guarantee session isolation and a formal correctness argument can be found in [22]. Informally, consider two clients C_1 and C_2 that compete for shared and exclusive access to R , respectively, and suppose that a shared-session request from C_1 gets accepted with $R.updateSID = \langle T_s^1, T_x^1 \rangle$ in its annotation. Observe that due to global uniqueness of session proposals, the owner of R would subsequently accept an exclusive-session request from C_2 with verifier $R.verifySID = \langle T_s^2, T_x^2 \rangle$ only if T_x^2 is strictly greater than T_x^1 . In this case, subsequent shared-session requests from C_1 would fail verification, causing C_1 to observe *EBADSESSION* and downgrade its lock. Thus, session isolation would be preserved in this example via a forced termination of C_1 's session. A similar argument demonstrates that no two exclusive-session commands can be interleaved by a conflicting command from another client.

3.4 Supporting distributed transactions

3.4.1 Overview and design requirements

Transactions are widely regarded as a useful programming primitive and traditionally, SAN-oriented applications implement transactional semantics using two-phase locking for isolation and a write-ahead logging (WAL) facility (sometimes referred to as *journaling*) for atomicity and durability. To support transactions, Minuet relies on these well-understood and widely-used mechanisms, while extending them with the use of the guard to address the safety problems outlined in Section 2.2. Since

the primary focus of this paper is feasibility of safe and highly-available applications in SANs rather than performance, we provide only a subset of features typically found in a state-of-the-art transaction service such as D-ARIES [38]. Below, we present a design that implements *Redo*-only logging to support the "no force no steal" buffer policy and currently, our design permits only one active transaction per process at a time - after starting a transaction, a client must commit or abort before initiating the next transaction. Finally, we assume unbounded log space for each client. These restrictions allow us to focus the discussion on the novel aspects of our approach and we believe that additional optimizations, such as support for *Undo* logging, can be easily retrofitted onto our scheme if necessary. The following set of requirements motivates our design:

(1) Avoid introducing assumptions of synchrony required by conventional transaction schemes for SAN environments. We rely on the guard at target devices to provide session isolation and protect the state on disk from the effects of arbitrarily-delayed I/O commands operating on the application data and the log.

(2) Eliminate reliance on strongly-consistent locking. Rather than requiring clients to coordinate concurrent activity via a strongly-consistent DLM, the guard at storage devices enables a limited form of isolation and permits us to relax the degree of consistency required from the lock service. Prior to committing a transaction, a client process in Minuet issues an extra disk request, which verifies the validity of all locks acquired at the start of the transaction. This mechanism allows us to identify and resolve cases of conflicting access due to inconsistent locking state at commit time and can be viewed as a variant of optimistic concurrency control - a well-known technique from the DBMS literature [30].

(3) Avoid enforcing a globally-consistent view of process liveness. Rather than relying on a group membership service to detect client failures and initiate log recovery proactively in response to perceived failures, our design explores a *lazy* approach to transaction recovery that postpones the recovery action until the affected data is accessed. This enables Minuet to operate without global agreement on group membership.

3.4.2 Basic transaction protocol

Minuet stores transaction *Redo* information in a set of per-client logs on shared disks. The physical location of a client's log can be computed from its client identifier (*clntID*). These logs appear to Minuet's transaction module as regular lockable resources that can be read from and written to, while the guard is assumed to enforce session isolation in the event of concurrent access from multiple clients.

To support transactions, we extend the basic session

isolation machinery described in Section 3.3 with an additional piece of state called a *commit session identifier* (*CSID*) of the form $\langle \text{clntID}, \text{xactID} \rangle$. We extend the format of a session annotation to include two commit session identifiers, denoted *verifyCSID* and *updateCSID*, and both are set to *NIL* unless specified otherwise. For each shared resource *R*, the owner device maintains a local commit session identifier (*R.ownerCSID*) as well as *R.ownerSID*. Conceptually, the value of *R.ownerCSID* at a particular point in time identifies the most recent transaction that may have updated *R* and committed without flushing its changes to the disk image of *R*. If *R.ownerCSID* \neq *NIL*, the current state of *R* on disk may be missing updates from a committed transaction and thus cannot be assumed valid. In this case, *R.ownerCSID.clntID* identifies the client process responsible for the latest transaction on *R* and it is used to locate the corresponding log for recovery purposes.

Upon receiving a disk request, the guard examines the annotation and rejects the request if *R.verifyCSID.clntID* \neq *R.ownerCSID.clntID* or if *R.verifyCSID.xactID* $<$ *R.ownerCSID.xactID*. A request is accepted *only if* its *verifySID* and *verifyCSID* both pass verification and upon completing the request, the owner device updates its local commit session identifier by setting *R.ownerCSID* \leftarrow *R.updateCSID*. If verification fails, the owner responds with *EBADSESSION* and attaches the tuple $\langle \text{R.ownerSID}, \text{R.ownerCSID} \rangle$ in a response annotation.

In Minuet, transactions proceed in five stages: *Begin*, *Read*, *Update*, *Prepare*, and *Commit* and we illustrate them using high-level pseudocode in [22]. During one-time client initialization, Minuet's transaction service locks the local client's log in *Excl* mode. To begin a new transaction *T*, the client selects a new transaction identifier (*curXactID*) via a monotonic local counter and appends a *BeginXact* record to its log. Next, in the *Read* phase of a transaction, the application process acquires a *Shared* lock on every resource in *T.readSet* and reads the corresponding data from shared disks into local memory buffers. In the *Update* phase that follows, the client acquires *Excl* locks on the elements of *T.writeSet*, applies the desired set of updates locally, and communicates a description of updates to Minuet's transaction service, which appends the corresponding set of *Update* records to the log. Each such record describes an atomic mutation on some resource in *T.writeSet* and essentially stores the parameters of a single disk *WRITE* command.

The *Prepare* phase serves a dual purpose: to verify the validity of client's sessions (and hence, the accuracy of cached data) and to lock the elements of the write set in preparation for committing. For each resource in *T.readSet* \cup *T.writeSet*, the client sends a special *PREPARE* request to its owner. Minuet im-

plements *PREPARE* requests as zero-length *READs*, whose sole purpose is to transport an annotation and invoke the guard. *PREPARE* requests for elements of *T.writeSet* carry *verifyCSID* = *NIL* and *updateCSID* = $\langle C, \text{curXactID} \rangle$ in their annotations, where *C* is the client's identifier. If all *PREPARE* requests return *SUCCESS*, the transaction enters the final *Commit* phase, in which a *CommitXact* record is force-appended to client *C*'s log.

The protocol outlined above ensures transaction isolation, identifying cases of conflicting access during the *Prepare* phase. Recall, however, that under the session isolation semantics, any I/O command, including operations on the log, may fail with *EBADSESSION* due to conflicting access from other clients. This gives rise to several exception cases at various stages of transaction execution. For example, a client *C* may receive an error while forcing *CommitXact* to disk due to loss of session to the log. This can happen only if another process has initiated log recovery on *C* and hence, the active transaction must be aborted. Other failure cases and the corresponding recovery logic are described in the report [22].

Syncing updates to disk: After committing a transaction, a client *C* flushes its locally-buffered updates to *R* simply by issuing the corresponding sequence of *WRITE* commands to its owner device. Each such command specifies in its annotation $\{ \text{R.verifyCSID}, \text{R.updateCSID} \} = \langle C, \text{syncXactID} \rangle$, where *syncXactID* denotes *C*'s most recent committed transaction that modified *R*. After flushing all committed updates, *C* issues an additional zero-length *WRITE* request, which specifies *R.verifyCSID* = $\langle C, \text{syncXactID} \rangle$ and *R.updateCSID* = *NIL* in the annotation. This request causes the device to reset *R.ownerCSID* to *NIL*, effectively marking the disk image of *R* as "clean". Lastly, *C* appends to its log an *UpdateSynced* record of the form $\langle R, \text{syncXactID} \rangle$.

Lazy transaction recovery: A client *C* can initiate transaction recovery when its disk command on some resource *R* fails with *EBADSESSION* and a non-*NIL* value *ownerCSID* = $\langle C_F, \text{xactID} \rangle$ is specified in the response annotation. This response indicates that the disk image of *R* may be missing updates from a transaction committed earlier by another client *C_F*. If *C* suspects that *C_F* has failed, it invokes a local recovery procedure that tries to repair the disk image of *R*. First, *C* acquires exclusive locks on *R* and *C_F.Log* and reads the log from disk. Next, *C* searches the log for the most recent transaction that has successfully flushed its updates to *R*, from which it determines the list of subsequent committed updates that may be missing from the disk image. The client then proceeds to repairing the state of *R* on disk by reapplying these updates and all *WRITE* requests sent to the owner during this phase specify

$\{R.verifyCSID, R.updateCSID\} = R.ownerCSID$ in the annotation. Finally, after reapplying all missing updates, C completes recovery by issuing a zero-length *WRITE* annotated with $R.verifyCSID = R.ownerCSID$, $R.updateCSID = NIL$. A more detailed discussion of transaction recovery in Minuet can be found in [22].

3.5 Lock manager replication

Some lock services seek to achieve fault tolerance by replicating lock managers. Since Minuet does not need to provide assurances of mutual exclusion, it relies on a simpler and more available replication scheme that permits clients to retain progress in the face of extensive node and connectivity failures. A lock can be acquired as long as at least one manager instance is reachable. In an extreme case, that instance can be the local Minuet client itself, which would simply grant its own proposals without coordinating with other processes.

To support manager replication, we extend the basic locking protocol presented in Section 3.3 as follows: When acquiring or upgrading a lock, a client selects a subset of managers, which we call its *voter set*, and sends an *UpgradeLock* request to all members of this set. The lock is considered granted once *UpgradeGranted* votes are collected from all members. If any of the voters respond with *UpgradeDenied* due to an outdated timestamp, the client downgrades the lock on all members that have responded with *UpgradeGranted*, updates its $maxT_s$ and $maxT_x$ values, and resubmits the upgrade request with a new timestamp proposal. As a performance optimization, we allow *UpgradeLock* requests to specify an *implicit downgrade* for an earlier timestamp.

4 Implementation

We have implemented a proof-of-concept prototype of Minuet based on the design presented in the preceding section. The prototype has been implemented on the Linux platform using C/C++ and consists of a client-side library, a lock manager process, an iSCSI protocol stack extension, and two sample parallel applications.

4.1 Core Minuet modules

Client-side library (5,440 LoC): The client-side component is implemented as a statically-linked library and provides an event-driven interface to Minuet's core services, which include locking, remote disk I/O, and transaction execution. When requesting a lock, a client can optionally specify the desired size of the voter set, which enables application developers to tune the degree of locking consistency, enabling a choice between optimism and strict coordination. A small voter set works well for low-contention resources; it helps keep the lock message overhead low and permits clients to make progress in a partitioned network. Conversely, a

large voter set requires connectivity to more manager replicas, but reduces the rate of I/O rejection under high contention. All outgoing disk commands are augmented with session annotations and in the event of rejection by the target device, a *ForcedDowngrade* event is posted to inform the application that the corresponding lock has been downgraded to some weaker mode.

Minuet lock manager (4,285 LoC): The lock manager process grants and revokes locks using the timestamp mechanism of Section 3.3 and several manager replicas can be deployed for fault tolerance. For each lockable resource, the manager maintains the current lock mode, the list of current holders, the queue of blocked upgrade requests, and the largest observed timestamp proposal.

SAN protocols and guard logic: To demonstrate the practicality of our approach, we implemented the guard logic and session annotations within the framework of iSCSI [4], a widely-used transport for IP-based SANs, and our prototype extends an existing software-based implementation of the iSCSI standard. On application client nodes, we modified the top and the bottom levels of the 3-tier Linux SCSI driver model. The top-level driver (*/drivers/scsi/sd.c*) presents the abstraction of a generic block device to the kernel and converts incoming block requests into SCSI commands. We extended *sd* with a new *ioctl* call, which enables the Minuet client library to specify session annotations for outgoing requests and to collect response annotations.

The bottom-level driver implements a TCP encapsulation of SCSI and our current prototype builds on the Open-iSCSI Initiator driver [6] v2.0-869.2. We used the *additional header segment* (AHS) feature of iSCSI to attach Minuet annotations to command PDUs and defined a new AHS type for this purpose.

Our storage backend is based on the iSCSI Enterprise Target driver [5] v0.4.16, which exposes a local block device to remote initiators via iSCSI. We extended it with the guard logic, which examines incoming PDUs and makes an accept/reject decision based on the annotation. Command rejection is signaled to the initiator via the REJECT PDU defined by the iSCSI standard.

The addition of guard logic represents the most substantial extension to the SAN protocol stack, but incurs only a modest increase in the overall complexity. The initial implementation of the Enterprise Target driver contained 14,341 lines of code and augmenting it with Minuet guard logic required adding 348 lines.

4.2 Sample applications

Distributed chunkmap (342 LoC): Our first application implements a read-modify-write operation on a distributed data structure comprised of a set of fixed-length

data chunks. It mimics atomic mutations to a distributed chunkmap - a common scenario in clustered middleware such as filesystems and databases. The chunkmap could represent a bitmap of free disk blocks, an array of i-node structures, or an array of directory file slots. In each iteration, the application selects a random chunk, reads it from shared disk, modifies a random chunk region, and writes it back to disk. To ensure update atomicity, the application acquires an *Excl* lock on the respective block from Minuet prior to reading it from disk and releases the lock after writing back the updated version.

Distributed B-Tree (3,345 LoC): To demonstrate the feasibility of serializable transactions, we implemented a distributed B-link tree [32] (a variant of B+ tree) on top of Minuet. Our implementation provides transactional *insert*, *delete*, *update*, and *search* operations based on the protocol presented in Section 3.4.2. For each operation, the application initiates a transaction and fetches the chain of tree blocks necessary for the operation (*Read* phase). Next, it upgrades the locks on the modified blocks to *Excl* mode and logs the updates (*Update* phase). Lastly, the client *Prepares* and *Commits* the transaction. If a transaction aborts due to loss of session to a tree block or the client’s log, the application reacquires the corresponding lock and retries (without back-off) until it commits successfully. For efficiency, clients retain *Shared* locks (and the content of cache buffers) across transactions and stale cache entries are detected and invalidated during the *Prepare* phase.

5 Evaluation

In this section, we evaluate the performance of our applications under different modes of locking. Due to space constraints, we present only key results that demonstrate the benefits of optimistic coordination enabled by Minuet and confirm the feasibility of our design. Several additional important measurements are reported in [22].

5.1 Experimental setup

For our experiments, we emulated a 39-node SAN environment interconnected via 100Mbps links using Emulab [41] and detailed hardware specifications are provided in Figure 4. Three of the nodes were configured to serve as Minuet lock managers and four additional nodes were used to emulate SAN-attached target devices, collectively providing 2GB of logical disk space, equally striped across the nodes. The remaining 32 nodes were configured as application clients. We ran each iteration of the experiment for 5 minutes and all of the values reported below are averages over 3 iterations.

In each iteration, we measure the aggregate *goodput* (the number of successful application-level operations

The number of storage targets	1	2	3	4
strong(1) coordination	105.0	220.0	329.9	412.4
strong(2) coordination	105.5	219.5	330.7	411.7
weak-own coordination	105.9	220.9	331.1	410.6

Table 1: Chunkmap application goodput (in operations per second) under the *uniform* workload.

per second) from all nodes and the rate of disk command rejection under the following locking configurations:

strong(x): We deploy a total of $2x - 1$ lock managers and require clients to obtain permissions from a majority (x). Note that *strong(1)* represents a traditional locking protocol with a single central lock manager, while *strong(2)* requires 3 lock manager replicas and masks one failure.

weak-own: An extreme form of weakly-consistent locking. Each client obtains permissions only from the local lock manager (co-located on the same machine) and does not attempt to coordinate with the other clients.

In all of our experiments, applications rely on Minuet to provide both modes of locking and do not make use of any other synchronization facilities.

5.2 Distributed chunkmap

In this experiment, we configured the chunk size to 8KB (for a total of 250K chunks) and ran the chunkmap application with 32 clients, varying the number of storage targets from 1 to 4. We considered two forms of workload:

uniform: In each operation, a chunk to be modified is selected uniformly at random.

hotspot(x): $x\%$ of operations touch a *hotspot* region of the chunkmap constituting 0.1% of the entire dataset.

Table 1 reports the aggregate goodput under the *uniform* workload, which represents a low-contention scenario. The goodput exhibits linear scaling with the number of storage servers. Further, there is no measurable difference in performance between the three locking configurations. These results suggest that the optimistic method of coordination enabled by Minuet does not adversely affect application performance, while providing safety, in scenarios where the overall I/O load is high, but contention for a single resource is relatively rare.

The rate of I/O rejection increases when the workload has hotspots and, as expected, *weak-own* suffers a performance hit proportional to the popularity of the hotspot (Figure 5). We note that the *hotspot* workload represents a very stressful case (the hotspot size is 0.1%) and our results demonstrate that weakly-consistent locking degrades *gracefully* and can still provide reasonable performance in such scenarios.

We also ran experiments in a partitioned network scenario, where each client can communicate with only a subset of replicas. A strongly-consistent locking protocol demands a well-connected primary component containing at least a majority of manager replicas - a condi-

	Storage targets	Lock managers	Clients
# Nodes	4	3	32
CPU	3GHz Xeon	850Mhz Pentium III	
RAM	2GB	512MB	
Disk	10K RPM SCSI	7200 RPM IDE	

Figure 4: Hardware specifications of the Emulab cluster.

	<i>tree-small</i>	<i>tree-large</i>
Node size	8KB	8KB
Fanout	150	150
Num. keys	187,500	18,750,000
Leaf occupancy	50%	50%
Tree depth	3	4
Total dataset size	20MB	2GB

Figure 6: Pre-populated B+ tree parameters.

tion that our partitioned scenario fails to satisfy. As a result, no client can make progress with traditional strong locking and the overall application goodput is zero. In contrast, under Minuet’s weak locking, clients can still make good progress. This demonstrates the availability benefits that Minuet gains over a traditional DLM design.

5.3 Distributed B+ tree

The B+ tree application demonstrates Minuet’s support for serializable transactions. In this experiment, we start with a pre-populated tree and run the application for 5 minutes on 32 client nodes. Each client inserts a series of randomly-generated keys and we measure the aggregate goodput, defined as the total rate of successful insertions per second from all clients. To test Minuet’s behavior under different transaction complexity and contention scenarios, we used two different pre-populated B+ trees, whose parameters are given in Figure 6.

Figure 7(left) compares the performance of *strong(1)* and *weak-own*. Under both locking schemes, the throughput exhibits near-linear scaling with the number of storage targets. As expected, *tree-large* demonstrates a lower aggregate transaction rate because each transaction requires accessing a longer chain of nodes. Moreover, since the number of leaf nodes is large, read-write or write-write contention is relatively infrequent and hence, the performance penalty due to I/O rejection incurred under *weak-own* is negligible. By contrast, *tree-small* represents a high-contention workload and our results suggest that even in this stressful scenario, Minuet’s weak locking incurs only a modest performance penalty.

Further investigation revealed that the primary cause of the performance degradation was an outdated estimate of maximum timestamps $\langle \max T_s, \max T_x \rangle$, causing

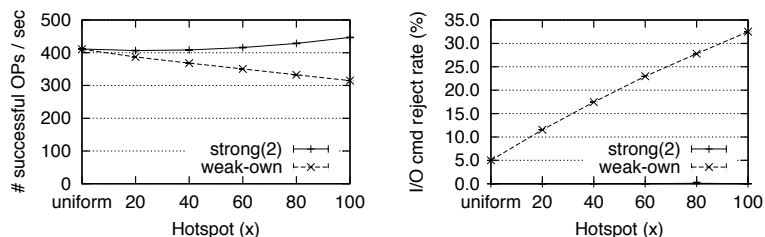


Figure 5: Left: chunkmap goodput under the *hotspot(x)* workload for varying x . Right: the rate of rejected I/O requests under *hotspot(x)* for varying x .

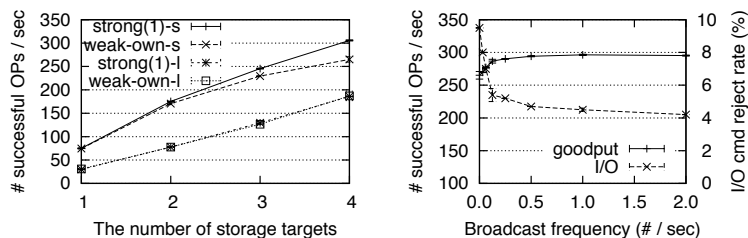


Figure 7: Left: B+ tree application goodput with *tree-small* and *tree-large* datasets. Right: effects of the timestamp broadcast optimization with *tree-small* dataset.

some of the commands to carry outdated session identifiers (e.g., with $\text{verifySID}.T_x < \text{ownerSID}.T_x$). Under *weak-own*, clients select session identifiers without coordinating with other clients and hence, a client may not know the up-to-date value of $\text{ownerSID}.T_x$ that may have been set by an earlier transaction from another client.

A simple optimization alleviating this issue is to let clients lazily synchronize their knowledge of maximum timestamps. More specifically, each client can broadcast its local updates on $\max T_s$ and $\max T_x$ to other clients at some fixed broadcast rate (b) and other clients can update their local estimates accordingly. We implemented this optimization and measured its effects on the *tree-small* workload with 4 storage targets. Figure 7(right) shows the results, which suggest that we can substantially reduce the rate of rejection by broadcasting with $b \geq 0.2$ and the resulting goodput closely approaches the maximum value achievable under strong locking.

Note that this optimization affects only performance and is not required for safety. Conceptually, the broadcast rate b provides a way of parameterizing the continuum between traditional locking and the fully optimistic case of *weak-own* and other methods may be possible.

6 Discussion

In this section, we discuss several issues pertaining to the practical feasibility of our approach and the implications of Minuet’s programming model.

Storage target modifications: Our approach rests on the basic idea of extending SAN-attached storage targets with a small amount of guard logic that enables them to detect and filter out inconsistent I/O requests, which will require storage array vendors to introduce a new feature into their products.

We acknowledge that Minuet relies on functionality that does not presently exist in standard storage hardware and, consequently, faces non-trivial barriers to standardization, implementation, and deployment. However, we observe that the proposed extensions are very incremental and can easily be retrofitted into an existing design. The guard logic is amenable to efficient implementation in hardware or firmware, requiring only a few table lookups and comparison operations.

As we argue above, the benefits of implementing such an extension can be substantial. In addition to lifting the safety and liveness limitations that have traditionally characterized shared-disk applications and middleware, our approach establishes a new degree of freedom in the design space of SAN applications, enabling a choice between optimism and strict coordination.

Our investigation builds upon earlier work on device locking [12], which has demonstrated the practical feasibility of this approach and the willingness of storage hardware vendors to adopt a promising new feature [2].

Metadata storage overhead: In our prototype implementation, target storage devices maintain 16 bytes of per-resource metadata. For a typical middleware service such as a database or a filesystem, a resource would correspond to a single fixed-length block containing application data or metadata and taking a clustered filesystem as an example, block sizes in the range 128KB - 1MB are common [8]. Assuming 128KB application block size, the table of Minuet session identifiers for a dataset of size 1TB would consume an additional 128MB.

Perhaps more alarmingly, Minuet metadata must be stored in random-access memory for efficient lookup on the data path. We envision the use of flash memory or battery-backed RAM for this purpose and observe that today, high-performance storage arrays make extensive use of NVRAM for asynchronous write caching [39]. Alternatively, the session state can be stored persistently on disk and a fixed-size NVRAM buffer can be used as a cache, providing efficient access to the working set.

Protocol extensions: Our approach requires augmenting the format of *READ* and *WRITE* commands with session annotations and our prototype implementation extends the iSCSI protocol with a new AHS type for this purpose. A transport-level modification simplified our software implementation, but would be difficult to deploy in a production environment, since it would require modifying the HBAs. For a more easily-deployable solution, the required set of extensions can be implemented in a transport-independent manner at the SCSI command level. One option would be to use an *extended command descriptor block (XCDB)*, as defined in SPC-4 ([13], section 4.3.4), and introduce a new descriptor extension type for carrying the session annotation. Likewise, command rejection can be signaled to the initiator via a *CHECK*

CONDITION status code with a new *additional sense code* and the response annotation can be communicated as *fixed-format sense data* ([13], section 4.5.3).

Programming model: Another concern is that Minuet imposes a different programming model, exposing application developers to additional exception cases that do not naturally arise under strong locking. When a traditional DLM service grants a lock to an application process, the lock is assumed to be valid and the client can proceed to accessing the disk without worrying about conflicting access from other clients. In contrast, Minuet gives out locks in a more permissive manner, but provides machinery for detecting and resolving conflicting access at the storage device. As a result, applications that rely on Minuet for concurrency control must be programmed with the assumption that any I/O request can fail with *EBADSESSION* due to inconsistent lock state.

We observe that while I/O rejection does not occur under conservative locking, the protocols employed by traditional DLMs for ensuring system-wide consistency of locking state inevitably expose application developers to analogous exception cases. For instance, a network connectivity problem causing some application node to lose connectivity to a majority of lock managers would typically cause that node to observe a DLM-related exception event. More concretely, the application process would be informed that due to lack of connectivity, some of its locks may no longer be valid - these are precisely the semantics of Minuet's *ForcedDowngrade* notification. Hence, both models demand exception-handling for dealing with forced lock revocation.

With Minuet, a node that finds itself partitioned from the rest of the cluster need not immediately give up its locks and instead, can perform a more granular recovery action. For example, it can switch to the optimistic method and resume disk access without coordinating with other application processes and this would permit it to make progress in the absence of conflicting access.

Our experience with developing and deploying sample applications on top of Minuet suggests that the availability benefits enabled by the use of such fine-grained recovery actions are certainly worth the extra implementation effort, which we believe to be relatively small. The chunkmap application was initially implemented on top of conventional locking using 327 lines of C code and extending the implementation to operate on top of Minuet required adding only 15 lines of code to handle the *EBADSESSION* notifications.

7 Related Work

Concurrency control has been extensively studied in the operating systems, distributed systems, and database communities. VMS [40] was among the first widely-available platforms to provide application developers

with the abstraction of a general-purpose distributed lock manager and today, DLMS are generally viewed as a useful building block for distributed applications.

Clustered and distributed filesystems [8, 10, 35–37] and relational databases [9] rely on locking or lease-based mechanisms to coordinate access to shared application state. Both sets of mechanisms make certain assumptions about timing, such as partially-synchronized clocks and bounded communication latency, in order to operate safely. These systems can directly leverage Minuet to ensure safe coordination of concurrent access to shared data on disk without assuming synchrony.

In web service data centers, distributed coordination services such as Chubby [20] and Zookeeper [15] have also become popular. These services are intended primarily for *coarse-grained* synchronization - a typical use case might be to elect a master among a set of candidates. Although the intended use of Minuet is to provide *fine-grained* synchronization in a shared-disk cluster, our system can also support such use cases by transitioning to strongly-consistent locking, whereby each lock is acquired with a majority voter set. Unlike our system, Chubby provides a hierarchical namespace and the ability to store small pieces of data, but these features are largely orthogonal to our approach. Chubby's *lock sequencer* mechanism allows servers to detect out-of-order requests submitted under an outdated lock and our timestamp-based *sessions* generalize this idea to support shared-exclusive locking. We also develop this notion further and observe that once we have the ability to reject inconsistent requests at the destination, very little is gained by enforcing strong consistency on replicated locking state and specifically, the use of an agreement protocol (e.g., Paxos [31]) may be more than necessary.

Concurrency control and transaction mechanisms have been extensively studied in databases. ARIES [33] is a state-of-the-art transaction recovery algorithm for a centralized database, supporting fine-granularity locking and partial rollbacks of transactions, while D-ARIES [38] extends this work to be usable in distributed shared-disk databases. Implementing these mechanisms on top of Minuet's locking and I/O facilities would ensure that they retain their safety properties in the face of arbitrary asynchrony. Minuet's basic transaction service presented in Section 3.4 is a variation of timestamp-based concurrency control - a standard and well-known technique in relational database design. Finally, database researchers have explored hybrid approaches to concurrency control [34] that enable tradeoffs between optimism and strict coordination and our work enables similar tradeoffs for general SAN applications, where the data resides on application-agnostic block devices.

There have been several research projects tackling the intelligence/information gap between operating systems

and storage systems [17, 25, 28]. These projects aim to achieve more expressive storage interfaces by exposing more information or adding more intelligence to storage devices. In our work, we identified and tackled safety problems in SANs by narrowing the intelligence gap between clustered applications and SAN storage devices.

Several earlier projects have investigated new approaches to concurrency control via functional extensions to storage devices. [29] proposes Dlocks as a new primitive for distributed mutual exclusion, whereby the lock acquisition state is maintained by the target devices themselves and manipulated by the initiators using a new SCSI command. Due to the inherent complexity of distributed locking, the lock management functionality has proven too difficult to implement in a SAN storage array and as a result, this mechanism did not attain wide acceptance among the storage device vendors. Follow-on work to the initial proposal presented a simplified scheme in form of DMEP [12]. In this scheme, storage devices expose an array of shared memory buffers holding the lock state and clients manipulate this state directly using simple atomic commands. The DMEP specification was implemented by a storage device vendor [2] and used by earlier versions of GFS [35].

Our work revisits the idea of device-assisted synchronization and is in line with these earlier efforts, but differs in several crucial respects. First, rather than extending the storage devices with lock management functions, we propose a more general synchronization primitive that supports a wider range of coordination techniques. In addition to "traditional" conservative locking, Minuet enables the use of optimistic concurrency control, which has been shown to reduce the synchronization overhead and deliver better performance for certain application workloads. As a result, Minuet enables a new degree of freedom in the design space of parallel SAN applications, enabling the developers to safely exploit the tradeoffs between synchronization overhead, access conflict rate, and application availability. Second, acquiring or releasing a lock in Minuet does not require explicit communication with the target storage device and instead, clients annotate outgoing I/O requests with the relevant synchronization state. This technique addresses the problem of delayed requests delivered under the protection of an outdated lock and thus enables SAN applications to guarantee safety despite arbitrary message delays, drifting clocks, and node failures. Finally, unlike prior proposals, our design does not require new SCSI commands and can be implemented within the confines of existing protocol standards.

Similar in spirit to this work, SCSI-3 Persistent Reserve [11] tries to address the safety problems in shared-disk environments by extending the storage protocol and target devices. Revoking a suspected node's reservation

typically necessitates a global decision on declaring the respective node faulty, which, in turn, requires majority agreement. Hence, SCSI-3 PR offers safety but not liveness in the presence of network partitions and massive node failures, while Minuet provides both.

8 Conclusion

This paper investigates a novel approach to concurrency control in SANs. Today, clustered SAN applications coordinate access to shared state on disks using strongly-consistent locking protocols, which are subject to safety and liveness issues in the presence of asynchrony and failures. To solve these problems, we augment target devices with a small amount of guard logic, which enables us to provide a property called session isolation and a relaxed model of locking which, in turn, provide a building block for distributed transactions. They also enable us to loosen the consistency requirements on distributed locking state, thus providing high availability despite failures and network partitions.

We have designed, implemented, and evaluated Minuet, a DLM-like synchronization and transaction module for SAN applications based on the protocols we presented. Our evaluation suggests that distributed applications built atop Minuet enjoy good performance and availability, while guaranteeing safety.

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments and our shepherd, James Bottomley, for his guidance.

References

- [1] Brocade 5300 switch. <http://www.brocade.com>.
- [2] Dot Hill and Sistina Software team up to improve computing performance for Linux users. http://findarticles.com/p/articles/mi_m0EIN/is_2001_August_28/ai_776015%52.
- [3] HP remote insight lights-out edition II (QuickSpecs). <http://www.hp.com>.
- [4] IETF RFC 3720: Internet small computer systems interface (iSCSI). <http://www.ietf.org>.
- [5] iSCSI enterprise target v0.4.16. <http://iscsitarget.sourceforge.net/>.
- [6] Open-iSCSI. <http://www.open-iscsi.org>.
- [7] OpenDLM. <http://opendlm.sourceforge.net>.
- [8] Oracle OCFS. <http://www.oracle.com>.
- [9] Oracle real application clusters. <http://www.oracle.com>.
- [10] Panasas PanFS. <http://www.panasas.com>.
- [11] SCSI-3 block commands (draft proposed standard). <http://www.t10.org>.
- [12] SCSI device memory export protocol (version 0.9.8). <http://www.t10.org>.
- [13] SCSI Primary Commands - 4 (SPC-4), working draft. project T10/1731-D revision 17. <http://www.t10.org>.
- [14] Survey finds SAN usage becoming mainstream. [http://findarticles.com/p/articles/mi_qa4137/is_200402/ai_n9362169/pg_1%](http://findarticles.com/p/articles/mi_qa4137/is_200402/ai_n9362169/pg_1%5).
- [15] ZooKeeper. <http://zookeeper.sourceforge.net>.
- [16] Private communication with IBM Research, 2005.
- [17] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *ASPLOS*, 1998.
- [18] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD*, pages 23–34, 1995.
- [19] T. Asaro. ESG analysis - the state of iSCSI-based IP SAN 2006. <http://www.netelligentgroup.com/articles/esgiscsi.pdf>.
- [20] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.
- [21] M. Carey, M. Franklin, and M. Zaharioudakis. *Fine-grained sharing in a page server OODBMS*. ACM, 1994.
- [22] A. Ermolinskiy, D. Moon, B.-G. Chun, and S. Shenker. Deploying and evaluating an iSCSI-based implementation of Minuet. In *UC Berkeley EECS TR*, Jan 2009.
- [23] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- [24] M. Franklin, M. Carey, and M. Livny. Transactional client-server cache consistency: alternatives and performance. *ACM TODS*, 22(3):315–363, 1997.
- [25] G. R. Ganger. Blurring the line between OSES and storage devices. In *CMU SCS Technical Report CMU-CS-01-166*, 2001.
- [26] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, 1989.
- [27] D. S. Hirschberg and J. B. Sinclair. Decentralized extremefinding in circular configurations of processors. *Commun. ACM*, 23(11):627–628, 1980.
- [28] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (IDISKs). In *SIGMOD Record*, Sept. 1998.
- [29] C. J. S. Kenneth W. Preslan, Steven R. Soltis, M. O’Keefe, G. Houlder, and J. Coomes. Device locks: mutual exclusion for storage area networks. In *IEEE MSS*, pages 262–274, 1999.
- [30] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *Readings in database systems (2nd ed.)*, pages 209–215, 1994.
- [31] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
- [32] P. L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. DB Syst.*, 6(4):650–670, 1981.
- [33] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1), 1992.
- [34] S. H. Phatak and B. R. Badrinath. Bounded locking for optimistic concurrency control. *Rutgers University TR DCS-TR-380*, 1999.
- [35] K. W. Preslan, A. Barry, J. Brassow, M. Declerck, A. J. Lewis, A. Manthei, B. Marzinski, E. Nygaard, S. V. Oort, D. Teigland, M. Tilstra, S. Whitehouse, and M. O’Keefe. Scalability and failure recovery in a linux cluster file system. In *USENIX ALS*, 2000.
- [36] O. Rodeh and A. Teperman. zFS - a scalable distributed file system using object disks. In *MSSD 2003*, pages 207–218.
- [37] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST 2002*, pages 231–244.
- [38] J. Speer and M. Kirchberg. D-ARIES: A distributed version of the ARIES recovery algorithm. *ADBIS Research Communi.*, 2005.
- [39] R. Treiber and J. Menon. Simulation study of cached RAID5 designs. In *HPCA*, page 186, Washington, DC, USA, 1995. IEEE.
- [40] J. W. E. Snaman and D. W. Thiel. The VAX/VMS distributed lock manager. *Digital Technical Journal*, Sept. 1987.
- [41] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI 2002*, pages 255–270.
- [42] M. Zaharioudakis, M. Carey, and M. Franklin. Adaptive, Fine-Grained Sharing in a Client-Server OODBMS: A Callback-Based Approach. *ACM Trans. on Database Syst.*, 22(4):570–627, 1997.

The USENIX Association

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

For more information about membership and its benefits, conferences, or publications, see <http://www.usenix.org>.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at <http://www.sage.org>.

Thanks to USENIX & SAGE Corporate Supporters

USENIX Patron

Microsoft®

Research

USENIX Benefactors

Google



Infosys®
POWERED BY INTELLECT
DRIVEN BY VALUES



USENIX & SAGE Partners

Ajava Systems, Inc.

DigiCert® SSL Certification

FOTO SEARCH Stock Footage
and Stock Photography

Splunk

Zenoss

USENIX Partners

Cambridge Computer Services, Inc.

GroundWork

Open Source Solutions

Xirrus

SAGE Partner

MSB Associates

